SublonK: Sublinear Prover PlonK

Arka Rai Choudhuri
NTT Research
arkarai.choudhuri@ntt-research.com

Sanjam Garg
UC Berkeley
sanjamg@berkeley.edu

Aarushi Goel
NTT Research
aarushi.goel@ntt-research.com

Sruthi Sekar*
IIT Bombay
sruthi.sekar1@gmail.com

Rohit Sinha
Swirlds Labs.
sinharo@gmail.com

ABSTRACT

We propose SublonK — a new succinct non-interactive argument of knowledge (SNARK). SublonK builds on PlonK [EPRINT'19], a popular state-of-the-art practical zkSNARK. Our new construction preserves all the great features of PlonK, i.e., it supports constant size proofs, constant time proof verification, a circuit-independent universal setup, and support for custom and lookup gates. Moreover, SublonK achieves improved prover running time over PlonK. In PlonK, the prover runtime grows with circuit size. Instead, in SublonK, the prover runtime grows with the size of the "active part" of the circuit. For instance, consider circuits encoding conditional execution, where only a fraction of the circuit is exercised by the input. For such circuits, the prover runtime in SublonK grows only with the exercised execution path.

As an example, consider the zkRollup circuit. This circuit involves executing one of n code segments k times. For this case, using PlonK involves proving a circuit of size $n \cdot k$ code segments. In SublonK, the prover costs are close to proving a PlonK proof for a circuit of size roughly k code segments. Concretely, based on our implementation, for parameter choices derived from rollup contracts on Ethereum, $n = 8, k = 128, s = 2^{16}$ (where s is the size of each code segment), the SublonK prover is approximately 4.8× faster than the PlonK prover. Proofs in SublonK are 2.4KB and can be verified in under 50ms.

KEYWORDS

table lookups, succinct non-interactive arguments of knowledge

1 INTRODUCTION

Succinct non-interactive arguments of knowledge (SNARK) [11, 53] are cryptographic primitives that allow a prover to generate a *small certificate* of correctness of a potentially expensive computation. Furthermore, these certificates are cheap to verify and can optionally hide secrets that the prover may have used in performing the computation (zero-knowledge SNARK). Over the past few years, realizing efficient SNARKs have been a topic of extensive research, including numerous applications (e.g., see [9, 33, 42, 58, 65, 68]) and real-world deployments (e.g., see [7, 25, 35]).

*Work done while at UC Berkeley.

Proceedings on Privacy Enhancing Technologies 2024(3), 314–335

https://doi.org/10.56553/popets-2024-0080

Constant Proof Size, Universal Setup SNARKs. An influential line of work on SNARKs [26, 34, 55] has focused on realizing them with constant proof size and constant time verification. In his seminal paper, Groth [35] gave a SNARK with a 3 group element proof. These short proof sizes make such SNARKs quite appealing for several applications, e.g., involving on-chain verification. However, these first-generation constant-size zkSNARK constructions relied on a circuit-specific trusted setup — making real-world deployments challenging. The next generation of constant-size SNARKs removed this obstacle by realizing universal setup SNARKs [36, 51]. In particular, these SNARKs relied on a circuit-independent trusted setup that only needed to be done once. Furthermore, following a circuit-specific pre-processing, these constructions also achieved constant time verification.

The PlonK Proof System. Improving upon prior work [51], Gabizon, Williamson, and Ciobotaru [25] introduced a practical universal setup SNARK with constant proof size (about 400 bytes in practice) and constant time verification. This construction has found widespread real-world deployments. A significant reason for the success of PlonK is its easy adaptability. For example, PlonK supports *custom gates* — gates other than + and × — that significantly enhance concrete performance. plookup [24] and PlonKup [56] augment PlonK to add support for *lookup gates* — a gate checking that its input is from a pre-defined *lookup list*.

Limitations of PlonK Proof System. A key limitation of PlonK, and all SNARKs with constant size proofs, is that proof generation is expensive — especially when compared to SNARKs with larger proof size, such as STARKs [7]. Thus, improving proof generation times for PlonK, or constant size SNARKs, continues to be an important problem of significant interest. A recent line of work on lookup arguments [20, 23, 57, 62, 64] improves PlonK proof generation for the special case of lookup gates to (essentially) independent of the lookup list. Targeting use cases where larger proof sizes are acceptable, several works [4, 16, 44] accelerate PlonK proof generation at the cost of increasing proof size.

Going Sublinear. We observe that PlonK and other similar SNARKs can be wasteful. For example, consider the following code snippet: *if X then evaluate* C_0 *else evaluate* C_1. When evaluating this code snippet, only the "active part," i.e., either C_0 or C_1, depending on the conditional X needs to be evaluated. In fact, it is often the case that in a computation, only a fraction of the circuit is "active."

More generally, we consider circuits C that can divided into $\bar{k}$ layers, where each layer has the same *branch* of n s-sized circuits $\mathsf{C}_1, \cdots, \mathsf{C}_n$. These layers are interleaved in C by activation layers

that select a single *active* circuit to execute in each layer - the choice of the circuit to be activated in each layer may depend on the input to C. Thus, for any input to C, the total size of the *executed/active* sub-circuit is $O(\bar{k}s)$ (independent of n). We denote circuits that satisfy the above structure as *layered branching circuit.*[1] In certain scenarios, the input to C may determine the total number of layers $k \leq \bar{k}$ that is activated - we handle this by adding a special identity circuit C_{id} (that passes input unchanged to the output), and requiring that for every layer $\geq k$, C_{id} is activated. For such inputs, we say that the *effective* number of layers is k. For the remainder of the introduction and overview, we will refer to the activated sub-circuit as *effective* activated sub-circuit of size $O(ks)$.

Rollups, for instance, naturally map to our model of computation, where at each step, the prover executes one of several transaction types. For example, in a typical decentralized exchange (DEX) smart contract (e.g., Loopring [1]), which allows users to create one of several types of transactions: deposits, spot trades, transfers, withdrawals, etc.

Naïvely, the prover runtime for SNARKs for the aforementioned circuits grows with the size of the entire circuit (i.e., grow with $O(nks)$). However, there has been a line of work [5, 8, 13, 15, 18, 27, 31, 40, 45–47, 49, 50, 52, 61, 67] addressing this issue by additionally requiring an "a la carte" cost profile from the prover, where the cost of proving should grow only with the size of the executed sub-circuit (of size $O(ks)$) rather than the entire circuit. Unfortunately, prior works either fail to achieve a constant proof size or resort to using cryptographic hash functions in a non-black-box manner[2], which is undesirable given the overhead caused.[3] See Section 1.4 for further discussion on this.

Specifically, our model allows the prover to perform a one-time pre-processing that depends on the circuit C. The stored pre-processed material may in fact depend on $O(nks)$, but the online proof generation cost grows only with the size of the *active* sub-circuit (i.e., $O(ks)$). Note that this pre-processing is separate from the pre-processing required for the verifier to achieve constant time verification.

1.1 Our Contributions

The current state of affairs thus motivates us to construct SNARKs for layered branching circuit in the pre-processing model where: (i) the proof is of constant size; and (ii) the overhead in the prover cost in selecting the active circuit at each layer is $O(1)$. Our contributions are as follows:

(1) We present SublonK, building on a popular SNARK system PlonK. Our new construction preserves all the great features of PlonK, i.e., it supports constant size proofs, constant time proof verification, a circuit-independent universal setup, and support for custom and lookup gates. Additionally, SublonK proof generation time grows only with the size of the active sub-circuit. Previously, PlonK proof generation grew with the size of the entire circuit.
(2) We provide an implementation of SublonK in Rust and evaluate it on circuits modeling a popular rollup application.
(3) We demonstrate the practical improvements with SublonK. For instance, in our rollup application, we demonstrate improvements in the prover time of up to 4.8× over the PlonK prover, and we also show potential far greater speedups for general-purpose programs that only exercise a small fraction of the entire logic in any execution. Proofs are 2.4KB in size, and verification requires 50 ms on a commodity machine or 716.6K EVM gas units to verify on-chain.

1.2 Our Techniques

We now discuss the key technical ideas underlying SublonK. As discussed earlier, our goal is to design pre-processing SNARKs for layered branching circuit, where the online cost to prove computation grows only with the active sub-circuit.

Recall that the pre-processing phase for the verifier in pre-processing SNARKs outputs a short summary - typically a commitment - to all the constraints in the circuit. These SNARKs then allow for the verifier to determine whether to accept the proof based on the commitment, thereby allowing the verifier to run in time sub-linear in the size of the circuit (i.e., the verifier no longer has to parse the entire circuit).

Core Idea: Note that once the prover executes the layered branching circuit C on input x, it induces the active sub-circuit $\widetilde{C}_x$ of size $O(ks)$ (with the non-active circuits from each layer removed). Thus, with the induced sub-circuit $\widetilde{C}_x$, the prover can generate the proof in time proportional to $O(ks)$ as desired. Unfortunately, such a proof is not useful to the verifier since it cannot verify the proof without a commitment to the constraints specified only by $\widetilde{C}_x$, whereas the verifier has a commitment to the constraints specified only by C. The first main idea is to enable the verifier to derive the commitment to $\widetilde{C}_x$ only given the commitment to C.

As a starting point, note that this is not something that can be addressed in the pre-processing phase since $\widetilde{C}_x$ depends on the input x. Further, having the prover simply send over the claimed commitment to $\widetilde{C}_x$ will not work either since the verifier needs to be convinced that the commitment was correctly generated. Thus we will require the prover to additionally prove that the new commitment is indeed a commitment to a valid induced sub-circuit of C, which the verifier can check given the commitment to C. But note that since the prover is now proving validity with respect to the original circuit of size $O(nks)$, care must be taken that the prover cost for this does not become proportional to $O(nks)$.

A natural approach for implementing this idea is to compute a Merkle hash of all the constraints in the pre-processing phase. Later in the online phase, the prover can generate a proof to convince the verifier that the new claimed commitment to $\widetilde{C}_x$ is a commitment to a subset of the leaf nodes (constraints) in this Merkle tree. This simple approach requires making a *non-black-box* use of hash functions, which adds significant computational overheads. The inefficiency of this type of approach is well established - in fact, a recent relevant line of work moved away from the non-black-box use

[1] While the applications we consider can naturally be cast into this model, our model is general enough to handle a larger class of circuits but may require more work to be viewed in this framework. Such transformations for more general circuits are orthogonal to our work.
[2] I.e., proving a statement about the hash function by representing it as a set of constraints.
[3] Non-black-box use of cryptography inherently induces a non-constant overhead when selecting the active circuit at each layer. More elaboration can be found in Section 1.4.

of hash functions for breakthrough results on efficient sub-linear time *lookup arguments* [20, 23, 57, 62, 64].

Lookup Arguments. As we shall see shortly, lookup arguments will, in fact, be central to our work, and here we provide an informal description of the requirements in such works. Specifically, given a table T of size N to which the verifier only has access via a pre-processed commitment[4], *lookup arguments* allow one to prove in time proportional to $\tilde{O}(m)$ (independent of N) that the values of a committed polynomial of size m are contained within the table T. This is achieved by allowing a one-time prover pre-processing on the table T, taking time proportional to $\tilde{O}(N)$, and re-usable across multiple proofs. The online proof generation time grows only with $\tilde{O}(m)$. In fact, the m values allow for *repeated* elements from the table T, a property we will crucially leverage. Finally, the verification for the lookup arguments we consider is, in fact, $O(1)$ time. The specific lookup protocol that we build on is *cached quotients* (or $\mathfrak{cq}$), introduced by Eagen, Fiore and Gabizon [20], which fits well with the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof system that we will use.

$\mathcal{S}\mathfrak{ublon}\mathcal{K}$ **Template:** Recall, that in the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof system[25], to achieve $O(1)$-time verification for a circuit C, there is an untrusted verifier pre-processing phase that outputs $O(1)$ sized commitment on input C. We refer to the pre-processed verifier commitment as a $\mathcal{P}\mathfrak{lon}\mathcal{K}$ commitment to C. Tying in with our previous discussion on *lookup arguments*, we have the following high-level template for $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ for layered branching circuit C: (i) generate a lookup table that *appropriately* encodes information about the layered branching circuit C - this will also require prover to compute a one-time prover pre-processing of the lookup table; (ii) once the induced circuit $\widetilde{\mathsf{C}}_x$ is fixed, use the lookup arguments to derive the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ commitment to C on-the-fly in time proportional to $\tilde{O}(ks)$ and prove that the derivation was done correctly. While this is indeed the template we follow in this work, there are several challenges in implementing its details that necessitate new ideas, as we illustrate below.

We begin by describing how we populate the initial table T given the layered branching circuit C in the context of $\mathfrak{cq}$. We use the implicit representation of C and store in T the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints for each circuit branch $\mathsf{C}_1, \cdots, \mathsf{C}_n$ [5] the exact nature of the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints are not important for this discussion. Since the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints for an s sized circuit can be represented in $O(s)$ constraints, the table consists of $O(ns)$ entries, where each entry is only a single field element (from an appropriate field).

In the online phase, once the induced circuit $\widetilde{\mathsf{C}}_x$ is fixed, the prover can compute the polynomial commitment com to the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints for $\widetilde{\mathsf{C}}_x$ as a concatenation of the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints for each active circuit in the k layers (since $\widetilde{\mathsf{C}}_x$ itself is a concatenation of k circuits)[6]. Since each of these constraints is present in the table T, the prover can simply run the lookup argument protocol in time proportional to $\tilde{O}(ks)$ to generate proof that the constraints are contained in T. Unfortunately, the only guarantee provided by $\mathfrak{cq}$, or any lookup argument, is that each element in the committed polynomial com is contained in the table T. While this is necessary, it is not a sufficient condition in our setting. For instance, the relative ordering of the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints is crucial for us to rely on the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ argument system since the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ security analysis assumes that the pre-processing is done correctly. This motivates us to extend the notion of lookup arguments to *segment* lookup arguments that we detail next.

Segment Lookup. To address our application's specific needs, we extend the $\mathfrak{cq}$ protocol to achieve a notion of *segment lookup*. The initial table T of size $O(ns)$ in a segment lookup protocol is sub-divided into n segments, each consisting of $O(s)$ contiguous elements in T (starting with the first element). The prover provides a commitment to a polynomial that encodes values in T and proves that the committed values additionally satisfy *segment granularity*. Specifically, for $O(ks)$ values committed to via the polynomial, each of the k segments of size $O(s)$ (starting with the first element) must correspond exactly to a segment in T, maintaining relative ordering with the segment. It is easy to see that if each segment corresponds to a $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraint, a segment lookup protocol will indeed provide the necessary guarantees to ensure that the polynomial commitment sent by the prover is, in fact, a $\mathcal{P}\mathfrak{lon}\mathcal{K}$ commitment to $\widetilde{\mathsf{C}}_x$ (for some x), where the validity of the choice of the segments will be checked separately by the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof system.

Unfortunately, we cannot use existing lookup protocols in a black-box manner to achieve segment lookup. We extend the ideas present in the $\mathfrak{cq}$ protocol to construct a new segment lookup argument, where the prover costs grow with the size of the polynomial that is committed.[7]

Putting It Together. The table T containing the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints for the circuit branches in C is pre-processed and provided to the verifier. Once the input x is fixed, the prover uses the induced circuit $\widetilde{\mathsf{C}}_x$ to compute the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ verifier pre-processing for $\widetilde{\mathsf{C}}_x$, and subsequently, the corresponding segment lookup proof for it, in time $\tilde{O}(ks)$. Thus, the $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ proof consists of (i) the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ verifier pre-processing for $\widetilde{\mathsf{C}}_x$; (ii) a segment lookup proof that the verifier pre-processing was correctly derived from T; and (iii) $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof for $\widetilde{\mathsf{C}}_x$ to be verifier using the verifier pre-processing sent by the prover. All the communication and verification can be done in $O(1)$, thus satisfying our efficiency requirements.

While this overview captures the main ideas, we refer the reader to the relevant technical sections for details. Specifically, we present our segment lookup protocol in Section 3 and show how it can be combined with $\mathcal{P}\mathfrak{lon}\mathcal{K}$ to get $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ in Section 4.

Remark 1. *As noted, our notion of segment lookup adds additional constraints to lookup arguments. These natural extension to lookup arguments are interesting in and of itself, and several extensions such as tuple lookup [19], matrix lookup [14] and index lookup [59] have already been considered in concurrent and subsequent works with*

[4] We do not need to make any trust assumptions about this pre-processing step since it can be recomputed and verified by anyone.

[5] We handle the activation layer constraints by embedding it within each circuit branch, such that the circuit branch activated in the j-th layer also outputs the circuit branch to be activated in the $j+1$-th layer, and thus the activation layer can be ignored for the purposes of our discussion.

[6] It should be noted that this description of a concatenation of $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints is not fully accurate and written here as such for simplicity, and we handle this in our technical sections.

[7] One could choose an appropriately large field to encode an entire segment into a single field element to use $\mathfrak{cq}$ in a black-box manner, but the overhead would be too large for this approach to be meaningful in our setting.

various applications (see Section 1.4 for details). One of our main insights is to identify that the segment lookup constructed extending cq *suffices to obtain a sublinear proof system for PlonK. In fact this observation was also made by a concurrent work [19] for the Marlin proof system. Refining this template to work for other (constrained) lookup arguments and proof systems is an exciting future direction.*

1.3 Example Applications

SublonK has the potential to improve prover run-time in nearly all applications of SNARKs, where the active part of the circuit during execution is not the entire circuit. This section explores some examples of applications where SublonK could be particularly beneficial and yield substantial computational savings.

(1) **Rollups:** Rollups are becoming increasingly popular due to their potential to address the scaling issue of modern layer 1 blockchains. Consider a typical decentralized exchange (DEX) smart contract (e.g. Loopring [1]), which allows users to create one of several types of transactions: deposits, spot trades, transfers, withdrawals, etc. The logic within these transaction types can be encoded as a circuit (typically under 60K arithmetic gates for each transaction). A single instance of a rollup transaction that is submitted to a layer 1 blockchain can batch together over hundreds of these DEX transactions, along with a single proof attesting to the validity of the state transition (from having applied all of the above DEX transactions on the state prior to the rollup transaction). Rollups naturally map to our model of computation, where at each step, the prover executes one of several transaction types (which map to segments in SublonK). Specifically, if there are n different DEX transaction types, and a rollup batches together k such DEX transactions (each of size s), then we expect SublonK to operate in roughly $O(ks\log(ks))$ time, whereas the PlonK prover would operate in $O(nks\log(nks))$ time. Our experiments in section 5 show significant speedups for rollups, for parameter values inspired by Loopring [1].
(2) **Smart Contracts:** Smart contracts support general computation (beyond rollups discussed above), but these can include arbitrary conditional statements, thereby often resulting in the active circuit only comprising a small fraction of the entire logic. For instance, consider a program that is a nested sequence of conditional statements - which can be represented as a complete tree in our graph-based model of computation. In such a setting, if each code segment is roughly the same size, the fraction of the executed path is exponentially smaller than the total size of the program.
Specifically in the above example, if the nested conditional statements resulted in $O(n)$ segments each of size s, the run time of a PlonK prover on any input would grow with $O(ns)$. However, since the execution path along the tree would only execute $O(\log(n))$ code segments, the SublonK prover running time would only grow with $O(\log(n)s)$. In section 5, we provide data points indicating significant concrete speedups for the above.
(3) **Proving Existence of Bugs in Large Codebase:** Exploitation attacks pose a significant risk to large and critical software systems, leading to the emergence of bug bounty programs. These programs involve independent research teams auditing deployed software and revealing vulnerabilities in exchange for monetary incentives.
Recent works [33, 38, 39] have explored the idea of using zero-knowledge proof systems as a means for vulnerability research teams to substantiate to bug bounty program managers that they have successfully detected a critical exploit. This guarantees that they obtain their reward without disclosing the exploit prematurely.
Although the relation circuit for these proofs grows with the size of the software system, the execution path needed to prove the existence of a bug is expected to be much smaller than the entire software system. Having the proof generation time depend on the size of the entire software system could be very costly, particularly for complex systems. For programs cast as layered branching circuit, SublonK is well-suited for such scenarios.
(4) **Combating Disinformation.** Naveh and Tromer [54] recently demonstrated that zero-knowledge proofs can be used to verify that images featured in media have undergone a pre-approved set of modifications since their creation. This capability is especially valuable as it helps journalists to hide sensitive content while simultaneously establishing the image's authenticity.
The complete list of pre-approved edits determines the size of the relation circuit for generating proofs. At the same time, the execution path only considers the edits that are applied to the image. SublonK could help significantly improve proof generation times in this application.

1.4 Related Works

Several prior works focus on building SNARKs where the prover cost grows only with the size of the program execution. However, all prior schemes (in the circuit-independent pre-processing model) either resort to non-black-box use of cryptography or do not achieve a constant proof size. We summarize the most relevant works in Figure 1 and give a detailed description below.

zkSNARKs for Disjunctions. Disjunctive statements are a special class of NP statements that comprise of a logical OR of a set of clauses. Building on the template introduced in Stacking-Sigmas [29], a recent work called Speed-Stacking [30] demonstrates how a large class of existing zkSNARKs can be modified to obtain zkSNARKs for disjunctive statements, where the prover work primarily grows with the size of the largest clause (with an additive overhead dependent on the total number of clauses). In particular, for a disjunctive statement consisting of ℓ clauses, each of size $|\mathsf{C}|$, the prover runtime in Speed-Stacking is $\tilde{O}(|\mathsf{C}| + \ell)$. The proof size has an additive overhead of $\log \ell$ over the proof size of the underlying zkSNARK. Thus, while fully black-box in the use of cryptography, Speed-Stacking leads to a non-constant proof size even if the underlying zkSNARK has a constant proof size.

A La Carte Cost Profile. There is a sequence of works including Buffet [61], vRAM [67] and Mirage [45] that consider an "a la carte" cost profile for the provers where the prover cost for proving a step of computation (akin to layers in our setting) grow only with the size of the circuit representing the instruction invoked on that step, i.e. independent of the number of branches. Mirage [45] achieves a constant proof size using the universal circuit approach, where the

trusted setup is run for the universal circuit (setup is indicated by a $*$ in Figure 1) and the executed circuit is passed as input to this universal circuit. Since the prover knows the executed sub-circuit, it can provide this input to the universal circuit. But to achieve constant verification time, one must pre-process the circuit passed to the universal circuit. Since the sub-circuit is input-dependent, it results in an input-dependent pre-processing to achieve an "a la carte" cost profile. vRAM [67] handles the issue of conveying the executed sub-circuit/instructions by only conveying the multiplicity of each instruction and appropriately encoding the constraints into the proof system, while fully black-box in the use of cryptography vRAM does not achieve a constant proof size. The proof generation in these works requires the entire transcript of the program execution in order to compute the proof, making them inherently non-incremental (see below).

Incremental Proofs. A recent line of work; Sangria [5], SuperNova [46], HyperNova [47], ProtoStar [13] address the lack of incremental property in the aforementioned works - incremental proof systems allow the prover to compute the proof alongside the computation, by simply performing a small update to the proof with each step of the computation. These works build on the novel folding technique introduced by Nova [48] for designing IVCs (incrementally verifiable computation). Sangria, SuperNova, HyperNova, and ProtoStar generalize the notion of IVCs to non-uniform IVCs, where at each step of the computation one out of a pre-determined set of instructions is executed. While the prover cost in these works only grows with the size of executed instructions at each step, they inherently rely on making non-black-box use of cryptography. This is because all these works follow the same high-level approach of designing an efficient folding argument and then efficiently compiling it into a non-uniform IVC using proof recursion. The proof size of the resulting non-uniform IVC depends on the underlying SNARK used in this compilation. In figure 1, we quote the proof sizes mentioned in the respective papers. However, we note that most of these schemes are compatible with and, hence can be adapted to work with various existing SNARKs (to reduce their proof size further).

Transparent Setup. The works on building zkSTARKs [8, 27, 31, 40, 49, 52] use a transparent (i.e. untrusted) setup. All these schemes use the algebraic intermediate representation (AIR), which only encodes the step-by-step trace of the program execution. This inherently leads to the a la carte prover cost since it will only grow with the AIR size, which grows only with the number of executed steps of the program. However, all these constructions have two shortcomings - non-constant proof size and non-black-box use of cryptographic hash functions. To ensure scalability while keeping the verifier pre-processing input independent, it is crucial that a hash of the computation trace is given along with proof that the hash computations were done correctly (uses AIR for "STARK-friendly" hashes). We summarize these properties for the most recent STARK [52] in figure 1.

Commit and Prove SNARKs (CP-SNARKs). Several works [15, 18, 50] build CP-SNARKs, which rely on proving statements of the form "$C_{ck}(w)$ contains w such that $R(x, w)$", where $C_{ck}(w)$ is a commitment. Such CP-SNARKs are shown to be useful [15] in proving the correctness of different parts of computation using different representations and proof systems (e.g., a QAP-based scheme may be used to prove one component, while a GKR [32]-based scheme may be used for another). LegoSNARK [15] builds a general framework for CP-SNARKs that would help in linking such different components and also build CP-SNARKs for some existing SNARKs (Groth [35], Pinnochio [60] and zk-vSQL [66]). This framework requires the prover to prove the knowledge of a valid opening for the commitment corresponding to the component used. Moreover, the only methods shown [18] to combine the proofs of different components involve a bounded bootstrapping (giving proof of a proof), making them non-black-box. We summarize these properties corresponding to the a la carte CP-SNARK, Geppetto [18], in figure 1.

Other Related Works. In a recent work [56], Pearson et al. also consider the idea of integrating $\mathcal{P}\mathfrak{lon}\mathcal{K}$ with a lookup argument [23]. However, their goal was very different from ours. They propose an extension of $\mathcal{P}\mathfrak{lon}\mathcal{K}$ that enables faster proof generation for relation circuits that include lookup gates without having to encode the lookup relation as an arithmetic circuit.

Concurrent Work. [19] In a concurrent work, Di et al. [19] build Mux-Marlin, which achieves similar efficiency parameters as our work (c.f. Figure 1) and follows a similar high-level technique as our work. While Mux-Marlin's segment lookup protocol (called "tuple lookup" in their paper) relies on the Plookup lookup argument [24] and combines it with the Marlin proof system [17], our $\mathcal{S}\mathfrak{ublon}\mathcal{K}$'s segment lookup uses cq lookup argument [21] and combines it with the Plonk proof system [25]. Since Mux-Marlin's segment lookup uses the Plookup lookup argument, its prover computation is $\tilde{O}((n+k)s)$. On the other hand, our use of $\mathfrak{cq}$ helps us get a prover computation of $\tilde{O}(ks)$. We remark that when the number of choices (n) is smaller than the number of execution steps (k), i.e., $n \leq k$, the prover time is asymptotically the same for both Mux-Marlin and $\mathcal{S}\mathfrak{ublon}\mathcal{K}$. On the other hand, for applications where $n \gg k$, $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ would give a better prover time.

Subsequent Works. In a subsequent work, Campanelli et al. [14] aims to generalize the kind of structure imposed by our segment lookup protocol on $\mathfrak{cq}$. In particular their generalization, which they call *matrix lookup*, is also able to improve upon the efficiency of the segment lookup protocol.

In a separate subsequent work, Setty, Thaler and Wahby [59] construct a new family of lookup arguments called Lasso. Of particular interest is their notion of an "indexed lookup argument" where a prover commits to two vectors of a and b, and proves that the i-th component of a is equal to b_i-th location of T, i.e. $a_i = T_{b_i}$. While similar to the notion of segment lookup, indexed lookup arguments do not suffice for our applications since we require additional (segment) structure on the "index vector" b, which is not enforced in Lasso. Further, from an efficiency perspective, the proof size of the indexed lookup arguments is *not* of constant size even if the underlying commitment used in their scheme supports constant size proofs of opening.

However, it should be noted that segment lookup arguments do *not* imply indexed lookup arguments since we leverage specific properties of the segment structure in our construction of the segment lookup argument. Thus, a natural open question is to extend

	Constant Proof Size	Input Independent Verifier Preprocessing	Black-Box in Cryptography	Incremental Proof	Setup
vRAM [67]	✗	✓	✓	✗	urs
Mirage [45]	✓	✗	✓	✗	urs*
Sangria [5]	✓	✓	✗	✓	urs
SuperNova [46]	✗	✓	✗	✓	trans
HyperNova [47]	✗	✓	✗	✓	trans
ProtoStar [13]	✗	✓	✗	✓	trans
eSTARK [52]	✗	✓	✗	✗	trans
Geppetto [18]	✗	✗	✗	✓	srs
Mux-Marlin [19] (C)	✓	✓	✓	✗	urs
Our Work	✓	✓	✓	✗	urs

Figure 1: Here, (C) refers to concurrent work. We use ✓ to denote that a certain property is satisfied and ✗ to denote that it is not. When we say that a scheme does not have constant proof size (by constant, we mean that the proof is a constant number of group or field elements), they have a size that depends on the program execution size. In the Setup column, we refer to a circuit-dependent setup by srs**, the circuit-independent universal setup by** urs**, and the untrusted transparent setup by** trans**. By** urs*, **we mean that while the setup is circuit-dependent, the particular scheme is defined for universal circuits.**

the ideas in this work to construct indexed lookup arguments, while still guaranteeing a constant proof size.

2 PRELIMINARIES

In this section, we present our model, establish notation and present an overview of $\mathcal{P}\mathfrak{lon}\mathcal{K}$. Due to space constraints, we defer additional preliminaries to Section A.

Notation. We denote our field by $\mathbb{F}$. We use $\mathbb{F}_{<d}[X]$ to denote the ring of univariate polynomials over $\mathbb{F}$ with a degree smaller than d. We denote our security parameter by λ. For a polynomial $P \in \mathbb{F}[X]$, and a subgroup $\mathbb{H} \subset \mathbb{F}$, we denote the evaluations of P at $\mathbb{H}$ by $P|_{\mathbb{H}}$. We use the additive notations for groups $\mathbb{G}_1$ and $\mathbb{G}_2$, and denote their corresponding group elements by $[x]_1 := x.g_1$ and $[x]_2 := x.g_2$, where g_1 and g_2 are the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. $[n]$ and $[k, n]$ are used to denote the sets of integers $\{1, \cdots, n\}$ and $\{k, \cdots, n\}$, respectively.

Lagrange, and Vanishing Polynomials. For a subgroup containing n-th roots of unity, i.e., $\mathbb{H} = \{1, \omega, \cdots, \omega^{n-1}\} \subset \mathbb{F}$, we denote the vanishing polynomial corresponding to $\mathbb{H}$ by $Z_{\mathbb{H}}(X) \in \mathbb{F}[X]$, defined as $Z_{\mathbb{H}}(X) := \prod_{i=1}^{n}(X - \omega^i)$. Furthermore, for each $i \in [n]$, we denote the i-th Lagrange polynomial corresponding to $\mathbb{H}$ by $\psi_i^{\mathbb{H}}(X) := \frac{Z_{\mathbb{H}}(X)}{Z'_{\mathbb{H}}(X-\omega^i)}$, where $Z'_{\mathbb{H}}$ is the derivative of the polynomial $Z_{\mathbb{H}}$.

Bilinear Groups. Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ be cyclic groups of prime order q with generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$. $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable and non-degenerate pairing, such that $e(h_1^\alpha, h_2^\beta) = e(h_1, h_2)^{\alpha\beta}$, for all $\alpha, \beta \in \mathbb{F}_q$, and all $h_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$.

2.1 Our Model

We discuss below the model of computation used in this work. At a high level, we will consider layered circuits C with $\bar{k}$ layers, where each layer supports a branch of n circuits $\{\mathsf{C}_1, \cdots, \mathsf{C}_n\}$. The circuits are connected via interleaving *activation layers* where the $j-1$-th activation layer specifies the circuit C_i that will be the *active branch* in the j-th layer. We allow the output of each activation layer to depend on the input to the circuit.

For simplicity, in our discussion we will "absorb" the aforementioned activation layer into the circuits such that the circuits can now be "concatenated". In particular, each circuit C_i now (i) also outputs the index of the next circuit to be activated; and (ii) has hardcoded the index i to check if it is indeed the circuit to be activated by a simple equality check of the hardcoded value and the incoming wires corresponding to the activated circuit. Thus, the concatenation of any two circuits $\mathsf{C}_i||\mathsf{C}_j$ simply states that the output wires of C_i contains the same values as the input wires of C_j.

Further, we note that our definition follows closely to that of works that consider (i) incremental verifiable computation (IVC) with non-uniform circuits at every step [13]; (ii) the "a la carte" cost profile of program execution [46, 61, 67].

We formally define the model below by specifying an efficiently computable function ξ that on circuit input specifies a vector $I \in [1, n]^{\bar{k}}$. For notational simplicity, we denote by ξ_x the vector $\xi(x)$, which is indexed at the j-th location by $\xi_x[j]$.

Definition 1. *Let $s, \bar{k}, n, m \in \mathbb{N}$, and for every $i \in [n]$, C_i is an arithmetic circuit of size s. Then, C is a $(s, \bar{k}, n, \{\mathsf{C}_i\}_{i=1}^n)$-layered branching circuit if C is such that there exists an efficiently computable function $\xi : \mathbb{F}^m \to [1, n]^{\bar{k}}$ such that for every input $x \in \mathbb{F}^m$, $\mathsf{C}(x) = \widetilde{\mathsf{C}}_\xi(x)$, where $\widetilde{\mathsf{C}}_\xi := \mathsf{C}_{\xi_x[1]}||\cdots||\mathsf{C}_{\xi_x[\bar{k}]}$.*

Remark 2. *To handle scenarios where the input determines the number of layers $k \le \bar{k}$ that are activated, we can include a 'special" circuit C_{id} which implements the identity (i.e., just passes the input through unchanged). We then say that for any $k \le \bar{k}$, the input x has* effective *layer k if the circuits active in the final $\bar{k} - k$ layers are*

all C_{id}, *i.e.* $\forall j \in [k, \bar{k}]$ $\mathsf{C}_{\xi_x[j]} = \mathsf{C}_{\mathsf{id}}$. *In such a situation, one can in fact require a stronger requirement - that the prover cost grows with* $O(ks)$ *rather than* $O(\bar{k}s)$. *For the rest of this work, we will indeed consider the effective layer* k.

The above formalism allows us to contrast between an explicit representation of the layered branching circuit, which is of size $O(kns)$, and the specific sub-sequence of k circuits each of size s that are *activated* for a specific input. Looking ahead, we will utilize the aforementioned property of layered circuits to construct a proof system where the prover cost grows with cost of the executed circuits $O(ks)$.

We note that while the example applications discussed in Section 1.3 can be naturally cast into this model, our model is general enough to handle a larger class of circuits. For instance, a generic program consisting of nested conditional statements of depth d can be recast in our above formulation by appropriately choosing $n = 2^d$ circuits, where the executed path will indeed consist of $d = O(\log n)$ circuits.

2.2 Background on $\mathcal{P}\mathtt{lon}\mathcal{K}$

$\mathcal{P}\mathtt{lon}\mathcal{K}$ [25] is a popular state-of-the-art, pre-processing zkSNARK with a constant-sized proof. As discussed earlier, our work builds on $\mathcal{P}\mathtt{lon}\mathcal{K}$ to design a zkSNARK where the proof generation time grows only with the size of the active sub-circuit. In this section, we provide the relevant background on the $\mathcal{P}\mathtt{lon}\mathcal{K}$ proof system that will be useful for understanding our construction. Some of the text in this section is taken verbatim from [25].

$\mathcal{P}\mathtt{lon}\mathcal{K}$ Constraint System. The $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraint system is meant to capture fan-in two arithmetic circuits of unlimited fan-out with n gates and m wires, but is more general. It is defined as $\mathscr{C} = (\mathcal{V}, \mathcal{Q})$, where:

- $\mathcal{V}$ is of the form $\mathcal{V} = (\mathbf{a}, \mathbf{b}, \mathbf{c}) \in [m]^{3n}$, which implicitly describe a *permutation* (to be explained shortly) on $[3n]$.
- $\mathcal{Q} = (\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C}) \in (\mathbb{F}^n)^5$ where $\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C} \in \mathbb{F}^n$ are the *selector vectors*.

We say $\mathbf{x} \in \mathbb{F}^m$ *satisfies* $\mathscr{C}$ if for each $\ell \in [n]$,

$$(\mathbf{q_L})_\ell \cdot \mathbf{x}_{\mathbf{a}_\ell} + (\mathbf{q_R})_\ell \cdot \mathbf{x}_{\mathbf{b}_\ell} + (\mathbf{q_O})_\ell \cdot \mathbf{x}_{\mathbf{c}_\ell} + (\mathbf{q_M})_\ell \cdot (\mathbf{x}_{\mathbf{a}_\ell} \mathbf{x}_{\mathbf{b}_\ell}) + (\mathbf{q_C})_\ell = 0.$$

This lets us define relation $\mathcal{R}_{\mathscr{C}}$ which is a set of pairs $\mathbf{x} := (\mathsf{x}, \mathsf{w})$, where $\mathbf{x}$ satisfies $\mathscr{C}$.

From Arithmetic Circuits to $\mathcal{P}\mathtt{lon}\mathcal{K}$ Constraint System. As an example, we demonstrate how a fan-in two arithmetic circuit with n gates (each one is either an addition or a multiplication gate) and m wires (since every gate is assumed to have 2 input wires and 1 output wire associated with it, $m = 3n$.) can be captured by the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraint system. For each gate $\ell \in [n]$

- Let $\mathbf{a}_\ell, \mathbf{b}_\ell$ and $\mathbf{c}_\ell$ denote the index of the left, right and output wire of the ℓ^{th} gate. Set $(\mathbf{q_C})_\ell = 0$. [8]
- Set $(\mathbf{q_L})_\ell = 0$, $(\mathbf{q_R})_\ell = 0$, $(\mathbf{q_O})_\ell = -1$, $(\mathbf{q_M})_\ell = 1$, when the ℓ^{th} gate is a multiplication gate.
- Set $(\mathbf{q_L})_\ell = 1$, $(\mathbf{q_R})_\ell = 1$, $(\mathbf{q_O})_\ell = -1$, $(\mathbf{q_M})_\ell = 0$, when the ℓ^{th} gate is an addition gate.

A circuit constraint system needs to ensure (1) *Correct Gate Evaluation:* given the left and right input wires, each gate is evaluated correctly. This is checked by choosing appropriate entries in the selector vectors based on the gate types (as described in the above example). (2) *Consistency of Wire Values:* if a wire is "split" (for instance as input to multiple gates or as the output of one gate and input to another), all the split wires must indeed contain the same wire value. This is done in the $\mathcal{P}\mathtt{lon}\mathcal{K}$ proof system via the *copy-check constraints* implemented by a permutation $\sigma : [3n] \to [3n]$. Specifically, σ is a collection of cycles (possibly of length 1), where each cycle is over all wires that are required to contain the same value as a consequence of the aforementioned "split".

We note that the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraint system can be further generalized to handle custom gates and gates with arbitrary fan-in. However, for simplicity of presentation, we only work with the above simple variant.

Verifier Pre-Processing in $\mathcal{P}\mathtt{lon}\mathcal{K}$. The $\mathcal{P}\mathtt{lon}\mathcal{K}$ protocol for the above constraint system is defined over a multiplicative subgroup $\mathbb{W} = \{1, \omega, \cdots, \omega^{n-1}\}$ of size n. Let k_1 and k_2 be picked such that $k_1 \cdot \mathbb{W}$ and $k_2 \cdot \mathbb{W}$ are distinct cosets of $\mathbb{W}$. [9] Let $\mathbb{W}' = \mathbb{W} \cup (k_1 \cdot \mathbb{W}) \cup (k_2 \cdot \mathbb{W})$. Identify $[3n]$ with $\mathbb{W}'$ via $\ell \to \omega^\ell$, $\ell + n \to k_1 \cdot \omega^\ell$, $\ell + 2n \to k_2 \cdot \omega^\ell$. Finally, let σ^* denote the mapping from $[3n]$ to $\mathbb{W}'$ derived from applying σ (as described above) and then this injective mapping into $\mathbb{W}'$.

The $\mathcal{P}\mathtt{lon}\mathcal{K}$ protocol requires the following universal trusted setup (needed for computing KZG commitments [43] throughout the protocol): $([\tau \cdot 1]_1, \ldots, [\tau^{n+5} \cdot 1]_1)$, for a randomly chosen τ. In addition, in order to keep the verifier cost low, the $\mathcal{P}\mathtt{lon}\mathcal{K}$ protocol pre-processes the constraint system to produce the following pre-processed input, where ψ_i correspond to the Lagrange polynomials over multiplicative sub-group $\mathbb{W}$.

$$n, (q_{M_\ell}, q_{L_\ell}, q_{R_\ell}, q_{O_\ell}, q_{C_\ell})_{\ell=1}^n, \sigma^*,$$

$$\mathsf{q_M}(X) = \sum_{\ell=1}^n q_{M_\ell}\psi_\ell(X), \mathsf{q_L}(X) = \sum_{\ell=1}^n q_{L_\ell}\psi_\ell(X),$$

$$\mathsf{q_R}(X) = \sum_{\ell=1}^n q_{R_\ell}\psi_\ell(X),\ \mathsf{q_O}(X) = \sum_{\ell=1}^n q_{O_\ell}\psi_\ell(X),$$

$$\mathsf{q_C}(X) = \sum_{\ell=1}^n q_{C_\ell}\psi_\ell(X), \mathsf{S}_{\sigma 1}(X) = \sum_{\ell=1}^n \sigma^*(i)\psi_i(X),$$

$$\mathsf{S}_{\sigma 2}(X) = \sum_{\ell=1}^n \sigma^*(n+\ell)\psi_\ell(X),\ \mathsf{S}_{\sigma 3}(X) = \sum_{\ell=1}^n \sigma^*(2n+\ell)\psi_\ell(X)$$

Here the KZG commitment to these polynomials are sent to the verifier as $[\mathsf{q_M}(\tau)]_1, [\mathsf{q_L}(\tau)]_1, [\mathsf{q_R}(\tau)]_1, [\mathsf{q_O}(\tau)]_1, [\mathsf{q_C}(\tau)]_1, [\mathsf{S}_{\sigma 1}(\tau)]_1, [\mathsf{S}_{\sigma 2}(\tau)]_1, [\mathsf{S}_{\sigma 3}(\tau)]_1$. The rest of the protocol proceeds by assuming that this pre-processing was done honestly, and that the verifier has access to these commitments, while the prover has access to the above polynomials. [10]

[8] We remark that the above is only an example. The $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraint system is quite general and can be used to enforce other types of constraints as well (e.g., checking if some wire value is equal to a public input by setting the corresponding entry in $\mathbf{q_C}$ to that public value, or whether a wire value is a boolean value etc.).

[9] Since further details regarding the cosets are not relevant to our discussion, we do not elaborate here and refer the reader to [25] for details.

[10] We omit the discussion on how $\mathcal{P}\mathtt{lon}\mathcal{K}$ works given the above pre-processing, since it is not relevant for understanding our techniques.

Observe that the size of the above polynomials (and hence the prover work in $\mathcal{P}\mathtt{lon}\mathcal{K}$) depends on the "entire" circuit. Looking ahead, we adopt the following high-level approach to reduce the prover work. Given "some" input-independent verifier pre-processing (based on the entire circuit), we will allow the prover to *efficiently derive* a "smaller" verifier pre-processing material (of the above form) that only depends on the activated sub-circuit and have the prover send KZG commitments to this derived pre-processing material to the verifier, along with a proof that certifies that these were computed honestly. Finally, given this "smaller" derived pre-processing, the rest of our protocol will work exactly as $\mathcal{P}\mathtt{lon}\mathcal{K}$. A majority of the rest of this paper is dedicated towards describing our approach that allows the prover to efficiently derive and convince the verifier that the derived verifier pre-processing for the activated sub-circuit was honestly computed.

3 SEGMENT-LOOKUP ARGUMENT

In this section, we present an efficient SNARK for *segment-lookup*. Looking ahead, in Section 4 we show how this protocol can be combined with $\mathcal{P}\mathtt{lon}\mathcal{K}$ to obtain $\mathcal{S}\mathtt{ublon}\mathcal{K}$.

3.1 Overview

We start with an overview of the techniques in our construction. As discussed in the introduction, the first step towards designing $\mathcal{S}\mathtt{ublon}\mathcal{K}$ involves designing a protocol for segment lookup. Since our protocol borrows techniques developed in cq[20], we first briefly recall the cq protocol. This will also allow us to pinpoint the shortcomings that necessitate a *segment* lookup protocol, which will allow us to naturally define the properties required from a segment lookup protocol.

Overview of cq [20]. Lookup arguments are succinct proof systems, where given a commitment to a large lookup table (of size n), the prover wants to convince the verifier that a commitment to a vector of k values are all contained in the large lookup table (the vector is allowed to repeat values from the table). cq allows the prover to pre-process the table and generate such proofs in time proportional to k, where the proofs themselves are constant-sized.

The lookup table in cq is encoded using a polynomial $T(X)$ of degree at most n, and the vector is encoded using a polynomial $F(X)$. Both these polynomials are committed via the KZG [43] polynomial-commitment. The pre-processing for the prover involves pre-computing succinct "quotient" commitments based on T. These quotient commitments help the prover generate a proof in the online phase in time that is proportional to $\tilde{O}(k)$.

In more detail, cq relies on the following log-derivative lemma from [37], which essentially says that the values encoded using polynomial $F(X)$ are a subset of the values encoded using polynomial $T(X)$ if and only if for some $m \in \mathbb{F}^n$

$$\sum_{i\in[n]} \frac{m_i}{X+t_i} = \sum_{i\in[k]} \frac{1}{X+f_i},$$

where t_i indicates the i-th entry of the table T, and correspondingly f_i indicates the i-th entry of F. Here m_i essentially encode the multiplicity of the i-th element T in F.

In [20], this identity is checked by letting the verifier evaluate it at a random β by requiring the prover to send polynomial commitments to the following polynomials: (1) $M(X)$, which is an encoding of m, and (2) polynomials $A(X)$ and $B(X)$, which are encodings of the summands on the left and right hand sides of the above equation respectively.

While $M(X)$ and $A(X)$ are both degree $n-1$ polynomials, the number of non-zero evaluations of these polynomials over $\mathbb{W}$ is at most k ($B(X)$ is only a sum of size $O(k)$). Hence, given pre-processed commitments to Lagrange polynomials, the prover can generate a commitment to M and A simply using $O(k)$ operations. The only remaining step in enabling the verifier to check the above equality is to convince them that the commitment to polynomial $A(X)$ is well-formed with respect to $T(X)$. This is be done by providing the verifier with a commitment to a quotient polynomial $Q_A(X)$ and letting them check if

$$A(X)\cdot(T(X)+\beta) - M(X) \stackrel{?}{=} Q_A(X)\cdot Z_{\mathbb{W}}(X).$$

Even though polynomials $A(X), M(X)$ have sparse representations, computing the $Q_A(X)$ still requires $O(n)$ operations. To reduce this overhead, cq introduced the idea of cached quotients. They show that if one pre-processes commitments to quotients $\{Q_i(X)\}_{i\in[n]}$, of the form

$$\psi_i^{\mathbb{W}}(X)\cdot T(X) = Q_i(X)\cdot Z_{\mathbb{W}}(X) + R_i(X),$$

then it is possible to compute a commitment to $Q_A(X)$ using just $O(k)$ operations.

As evidenced from the above equations, the cq protocol only guarantees that the elements encoded via $F(X)$ are contained in $T(X)$. As described in the introduction, our application for the lookup protocol requires stronger properties, as described below.

Segment-Lookup. We propose a variant of the standard lookup problem called the *segment-lookup* problem. At a high level, the look-up table of size ns is partitioned into n segments consisting of s elements each. Now the polynomial $F(X)$ encoding ks elements must contain k segments from $T(X)$ in its entirety, ensuring that the relative order of elements within each segment is maintained. From our aforementioned discussion of cq, it does not guarantee such a property. We formally define our requirements now.

Let $\mathbb{W} = \{\omega^0, \ldots, \omega^{ns-1}\}$ be a set of ns^{th} roots of unity and $\mathbb{V} = \{\nu^0, \ldots, \nu^{ks-1}\}$ denote the set of ks^{th} roots of unity. Given commitments to $ns-1$ degree polynomial $T(X)$, for each $i \in [0, n-1]$, the i-th segment is $\{T(\omega^{is}), \cdots, T(\omega^{(i+1)s-1})\}$. Thus, given a degree $ks-1$ polynomial $F(X)$, the prover in a segment-lookup protocol wants to convince the verifier that for each $i \in [0, k-1]$, $F(\nu^{is+0}), \ldots, F(\nu^{(i+1)s-1})$ represents one of the original n segments in $T(X)$. In other words, the prover wants to prove that $F(X)$ is well-formed, and each segment embedded in $F(X)$ is taken from $T(X)$.

Considering the above requirement, we want to design an efficient pre-processing SNARK with a constant proof size and constant time verification, where the prover work only grows with ks.

Our Approach. We now describe our main ideas for designing such a protocol. For simplicity of notation, for any polynomial $P(X)$, we will denote by p_i the evaluation at ω^i (the i-th power of the corresponding set of roots of unity), i.e., $p_i := P(\omega^i)$. Note that the

range of i will vary depending on the corresponding roots of unity P is defined over.

Similar to cq, we start by defining a polynomial $M(X)$ that is used to indicate which elements of $T(X)$ are included in $F(X)$ and how many times. i.e. m_i = #times t_i appears in $(f_1, \cdots, f_{ks})$. The number of non-zero entries in M is bounded above by $\min\{ks, ns\}$.

Since we want the "granularity" of elements selected to correspond to an *entire segment*, unlike cq, we want to enforce additional constraints on $M(X)$.

- **Constraint I:** The first constraint we enforce is that the value $\{m_i\}_{i\in[js,(j+1)s-1]}$ in $M(X)$ corresponding to each segment, must all be *equal*. We capture this by comparing each pair of consecutive values in M, except for the first value in each segment, since each can have distinct values. The test is then described as, $\forall i \in [ns]$ s.t. $s \nmid i, m_i = m_{i-1}$. Or, in polynomial terms,

$$\forall x \in \mathbb{W},\ (x^n - 1)(M(x) - M(x/\omega)) = 0.$$

  By the fact that $\mathbb{W}$ consists of ks-th roots of unity, first term, $(x^n - 1)$ is 0 if and only if $x = \omega^{js}$ for some j, which ensures: (i) the equation is trivially satisfied by the starting index of each segment, which by description is of the form js, i.e. encodes $s \nmid i$; and (ii) for all other indices i, it must be the case that $m_i = m_{i-1}$ to ensure the condition holds. This check is encoded by the polynomial check in Step 2 of Round 1.
- **Constraint II:** A consequence of how the table lookup protocol works in cq is that the relative ordering of elements in $F(X)$ need not be consistent with the relative ordering of elements in $T(X)$. Since our segment will encode a circuit, it is imperative that we maintain the relative ordering of the elements *within* a segment. We encode this test by first defining a function, $L : [ks] \mapsto [ns]$ which we will encode as a polynomial $L(X) : \mathbb{K} \mapsto \mathbb{W}$ (by overloading notation).[11] L maps the indices from F to their corresponding location within T. For relative ordering, we perform a check akin to that of M. The indices for consecutive elements in F should be consecutive in T (except for the start of the segment), $\forall i \in [ks]$ s.t. $s \nmid i, \ell_{i+1} = \ell_i + 1$. In polynomial terms, since we have ks-th roots of unity, we can rewrite this as,

$$\forall x \in \mathbb{V},\ (x^k - 1)(L(x\nu) - \omega L(x)) = 0.$$

  $L(X)$ has degree at most ks.
- **Constraint III:** Unfortunately, the above checks are not yet sufficient to achieve the desired segment granularity. While the tests do ensure that relative ordering is maintained within the segment, it does not enforce that the segments in $F(X)$ indeed *start* at the specified location, i.e. we want to ensure that ℓ_{js} for each $j \in [k]$ must map to *an* index *is* for some $i \in [n]$ in T. This will ensure that each segment in F indeed corresponds to a segment in T.
  We define a polynomial D, that selects $\{\ell_{js}\}_j$, i.e. $\forall i \in [k], d_{is} = \ell_{is}$. Finally, we need to check that $\forall i \in [k], s \mid d_{is}$. In polynomial terms, this translates to checking if all of the elements in $\{d_{is}\}_{i\in[k]}$ are n-th roots of unity. To perform this check, we invoke a sub-protocol from [62].

[11] Since the same segment can be invoked multiple times, the function is not injective, and thus L^{-1} is not well defined.

Most of the remaining protocol follows the cq template, except that since we need to enforce the above additional constraints, the log-derivative lemma used in cq does not suffice in our setting. We work with the following modified lemma: Each segment embedded in F is taken from T if and only if for some M and L as defined above,

$$\sum_{i\in[ns]} \frac{m_i}{X + t_i + Y\omega^i} = \sum_{i\in[ks]} \frac{1}{X + f_i + Y\ell_i}$$

The verifier picks random field elements to replace X, Y. The prover provides commitments to polynomials $A(X)$ and $B(X)$ that encode the summands in the LHS and RHS, respectively. Similar to cq, the verifier then verifies if the above equality holds. Throughout the protocol, when computing commitments to quotient polynomials of degree $ns - 1$, we use the idea of cached quotients to reduce online prover work at the cost of some additional pre-processing.

3.2 Definition

We begin by formally defining Segment-Lookup, secure against polynomial time adversaries.

Definition 2 (Segment-Lookup). *(n, k, s)-Segment-Lookup is a pair* $(\mathsf{gen}, \mathsf{segmentLookup})$ *such that:*

- $\mathsf{gen}(n, k, s, T)$: *This is a PPT algorithm that takes as input integers n, k, s and a polynomial $T \in \mathbb{F}[X]$ of degree $ns - 1$. It outputs a string* srs *of $\mathbb{G}_1$ and $\mathbb{G}_2$ elements. For a random $\tau \in \mathbb{F}$ and for* $\mathsf{max} = \max(k, n)$, srs *consists of $\{[\tau^i]_1, [\tau^i]_2\}_{i\in[0,\mathsf{max}\cdot s-1]}$, and other group and field elements. This algorithm is run in the pre-processing phase.*
- $\mathsf{segmentLookup}(\mathsf{com}, \mathsf{srs}, T, F, \mathbb{V})$: *This is an interactive public coin protocol between the Prover and the Verifier, where the prover has a private input $F \in \mathbb{F}[X]$ of degree $ks - 1$, and both the parties have access to T,* com *and* $\mathsf{srs} := \mathsf{gen}(n, k, s, T)$*, such that it satisfies the following properties of completeness and knowledge soundness in the algebraic group model. Let $\mathbb{W} = \{\omega^0, \ldots, \omega^{ns-1}\}$ be a set of ns^{th} roots of unity and $\mathbb{V} = \{\nu^0, \ldots, \nu^{ks-1}\}$ denote the set of ks^{th} roots of unity.*
  - ***Completeness.*** *If* $\mathsf{com} = [F(\tau)]_1$ *(τ as in the* srs*), and if for each $i \in [0, k-1]$, there exists $j \in [0, n-1]$ such that, for each $q \in [0, s-1]$, $F(\nu^{is+q}) = T(\omega^{js+q})$, then the verfier accepts with probability* 1.
  - ***Knowledge Soundness.*** *The probability of any* $\mathsf{poly}(\lambda)$*-time adversary $\mathcal{A}$ winning the following game is* $\mathsf{negl}(\lambda)$.
    (1) $\mathcal{A}$ chooses integer parameters n, k, s and the polynomial $T(X) \in \mathbb{F}[X]$ of degree $ns - 1$.
    (2) Compute $\mathsf{srs} := \mathsf{gen}(n, k, s, T)$.
    (3) $\mathcal{A}$ sends a message com *and polynomial $F(X) \in \mathbb{F}_{<ks}[X]$ such that* $\mathsf{com} = [F(\tau)]_1$*. Here, for* $\mathsf{max} = \max(k, n)$*, all $\mathbb{G}_1$ elements in the* srs *are linear combinations of $\{[\tau^i]_1\}_{i\in[0,\mathsf{max}\cdot s-1]}$.*
    (4) $\mathcal{A}$ (as the prover) and the verifier run the interactive protocol $\mathsf{segmentLookup}(\mathsf{com}, \mathsf{srs}, T, F, \mathbb{V})$*, where $\mathbb{V} \subset \mathbb{F}$ is a subgroup of order ks generated by ν.*
    (5) $\mathcal{A}$ wins if and only if the verifier accepts and there exists some $i \in [0, k-1]$, such that for each $j \in [0, n-1]$ there exists at least one $q \in [0, s-1]$ such that $F(\nu^{is+q}) \neq T(\omega^{js+q})$.

3.3 Protocol

Due to space constraints, we defer a formal description of our segment lookup protocol to Section B.2 and prove the following theorem in Section B.3. The building blocks and instantiations are described in Section B.1. In Section B.4 we discuss how our protocol can be easily augmented to achieve zero-knowledge with the caveat that the effective number of layers k is leaked to the verifier.

Theorem 1. *Let n, k, s be integers. Assuming that the $(\max(ks, ns))$-DLOG, qSDH, qDHE, and qSFrac assumptions [12, 28] hold, the aforementioned* (gen, segmentLookup) *is a (k, n, s)-segment-lookup protocol (see Definition 2) under the algebraic group model and random oracle model. The* gen *algorithm requires $O((\max \cdot s) + ns \log(ns))$ $\mathbb{G}_1$- and $\mathbb{F}$-operations, and $O(\max \cdot s)$ $\mathbb{G}_2$-operations. The prover computation cost in* segmentLookup *is $O(ks \cdot (\log ks + \log n))$ $\mathbb{G}_1$- and $\mathbb{F}$-operations and the proof size and verifier time are both $O(1)$.*

A detailed analysis of the prover cost of our segment-lookup protocol can be found in Appendix B.3.

4 $\mathcal{S}\mathtt{ublon}\mathcal{K}$: SEGMENT LOOKUP + $\mathcal{P}\mathtt{lon}\mathcal{K}$

In this section, we will utilize the segment-lookup protocol described in Section 3 in conjunction with the $\mathcal{P}\mathtt{lon}\mathcal{K}$ proof system to construct a proof system where, as described in the introduction, the prover cost grows with the length of the execution path, rather than the entire circuit. We elaborate on this requirement now. Consider a layered branching circuit C with at most $\bar{k}$ layers, where each layer has the same branch of n s-sized circuits $\mathsf{C}_1, \ldots \mathsf{C}_n$. On a given input, let it be the case that a sequence of $k \leq \bar{k}$ circuit branches $\widetilde{\mathsf{C}} := (\widetilde{\mathsf{C}}^{(i)})_{i \in [k]}$ (where k, as well as the sequence, is possibly dependent on the input), are executed in order. We want the prover cost to grow with the size of the total *executed* sub-circuit of size ks, rather than kns. For simplicity of notation, throughout the rest of this section, we will assume that the effective number of layers k is fixed in advance (and hence even the pre-processing algorithm will take k as input). At the end of Section C.2, we discuss how our protocol can be generalized to handle any $k \leq \bar{k}$ and hence the pre-processing of $\mathcal{S}\mathtt{ublon}\mathcal{K}$ only needs to take $\bar{k}$ as input.

Our solution's core idea is to store the encoding of the circuit constraints for each sub-circuit in a segment-lookup table. Given the input x, the prover uses the induced circuit $\widetilde{\mathsf{C}}_x$ to compute the $\mathcal{P}\mathtt{lon}\mathcal{K}$ verifier pre-processing for $\widetilde{\mathsf{C}}_x$, and subsequently, the corresponding segment lookup proof for it, in time $\tilde{O}(ks)$. In more detail, as discussed in Section 2.2, the $\mathcal{P}\mathtt{lon}\mathcal{K}$ proof system encodes circuit constraints via a number of polynomials that are pre-processed to their KZG commitment (see Section A.2) and provided to the verifier. Since the output of our segment-lookup protocol in Section 3 is indeed a KZG commitment to the polynomial defined by the selected segments, this gives the following natural approach to achieving a sub-linear prover:

(1) We represent each of the circuit branches $\mathsf{C}_1, \ldots, \mathsf{C}_n$ using the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraint system. Generate a table T where the i-th segment contains the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraints for C_i, i.e. T consists of n segments.
(2) Depending on the sequence of activated circuit branches, the prover can use our segment-lookup protocol (from Section 3) to generate the KZG commitment of the k constraints corresponding to the k activated circuit branches along with a proof of correctness.
(3) The prover post-processes the above derived KZG commitment so that it has the same form as the KZG commitments received by a $\mathcal{P}\mathtt{lon}\mathcal{K}$ verifier in the pre-processing phase for $\widetilde{\mathsf{C}}$.
(4) Given the above, we can directly rely on the $\mathcal{P}\mathtt{lon}\mathcal{K}$ protocol for $\widetilde{\mathsf{C}}$.

Unfortunately, while this is indeed the template for our protocol, the details do not work out in such a straightforward manner. We present the details in this section.

4.1 Pre-Processing Layered Branching Circuit

In this section, we describe how the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraints for each of the circuit branches $\mathsf{C}_1, \ldots, \mathsf{C}_n$ is stored in a table T consisting of n segments. We start by establishing some notation.

Ensuring Correct Sequence of Activated Circuit Branches. The *activated sub-circuit* $\widetilde{\mathsf{C}}$ (of C on input x) of size ks is specified by a sequence of k activated circuit branches $(\widetilde{\mathsf{C}}^{(i)})_{i \in [k]}$. Further, as discussed in Section 2.1, $\exists$ a function $\xi : \mathbb{F}^m \to [1, n]^k$ such that $\forall i \in [k]$, $\widetilde{\mathsf{C}}^{(i)} = \mathsf{C}_{\xi_x(i)}$ (where we use ξ_x to denote ξ_x). We assume that each circuit branch when executed, specifies the next circuit branch to be executed in the next layer. To ensure that the circuit branches are executed in order, we assume that one of the output wires for $\widetilde{\mathsf{C}}^{(i)}$ outputs $\xi_x(i+1)$, and each circuit $\mathsf{C}^{(i)}$ has hardcoded within it the index i to check whether the aforementioned incoming wire into $\xi_x(i+1)$ indeed has the hardwired value $\xi_x(i+1)$.

$\mathcal{P}\mathtt{lon}\mathcal{K}$ constraints. Let us denote the n $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraint systems for the n *circuit branches* $\{\mathsf{C}_i\}_{i \in [n]}$ as $\{\mathscr{C}^{(i)} = (\mathcal{V}^{(i)}, \mathcal{Q}^{(i)})\}_{i \in [n]}$. The corresponding pre-processed polynomials (refer to Section 2.2 for notation) are denoted by $(\mathsf{q_M}^{(i)}(X), \mathsf{q_L}^{(i)}(X), \mathsf{q_R}^{(i)}(X), \mathsf{q_O}^{(i)}(X), \mathsf{q_C}^{(i)}(X), \mathsf{S}_{\sigma_1}^{(i)}(X), \mathsf{S}_{\sigma_2}^{(i)}(X), \mathsf{S}_{\sigma_3}^{(i)}(X))$. These polynomials will be relevant to our subsequent discussion.

Let $\{\mathbf{x}_\ell\}_{\ell \in [3ks]}$ be the extended witness/satisfying assigment for the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraints such that for each $j \in [k]$, $\{\mathbf{x}_{js+i}, \mathbf{x}_{ks+js+i}, \mathbf{x}_{2ks+js+i}\}_{i \in [s]}$ correspond to the extended-witnesses for branching circuits $\widetilde{\mathsf{C}}^{(j)} (= \mathsf{C}^{(\xi_x(j))})$.

Checking Wire-Consistency or Copy-Check Constraints. Recall from Section 2.2, that the consistency of wire values in a circuit is checked in $\mathcal{P}\mathtt{lon}\mathcal{K}$ using the copy-check constraints. In our setting, in addition to checking these constraints within each individual circuit branch, we need an additional copy-constraint. In particular, in our setting, the j-th output wire of a circuit $\widetilde{\mathsf{C}}^{(i)}$ must have the same value as the j-th input wire of $\widetilde{\mathsf{C}}^{(i+1)}$. We ensure this by assuming that the output wires of $\widetilde{\mathsf{C}}^{(i)}$ are *not* a part of any cycles induced by the copy-check constraints $\sigma^{(i)}$ within $\widetilde{\mathsf{C}}^{(i)}$. For instance, this requirement can be satisfied by considering "output gates" in the final layer of the circuit that pass values unchanged, but ensure the input and output wire remain in different cycles within the permutation. Finally, since the (relative) index of the j-th output wire of $\widetilde{\mathsf{C}}^{(i)}$ is fixed and known to $\widetilde{\mathsf{C}}^{(i+1)}$, the copy-constraint for $\widetilde{\mathsf{C}}^{(i+1)}$ is constructed ensuring that the j-th output

wire of $\widetilde{\mathsf{C}}^{(i)}$ and j-th input wire of $\widetilde{\mathsf{C}}^{(i+1)}$ are a part of the same cycle in $\sigma^{(i+1)}$. Thus in each circuit branch C_i, if there are p input and output wires, the permutation within the circuit-branch is defined to be $\sigma^{(i)} : [-3p : 3(s-p)] \to [-3p : 3(s-p)]$.

This avoids any potential for cycles across circuit branches, except for the ones we discussed, and thus $\widetilde{\mathsf{C}}$ consists of disjoint cycles, and thus $\{\sigma^{(i)}\}_{i\in[n]}$ remains a permutation. For simplicity of notation, for the remainder of the section, we treat $\sigma^{(i)}$ to be $\sigma^{(i)} : [3s] \to [3s]$.

Parallel Repeated Version of the Segment-Lookup Protocol. Before the next discussion, we remark that in our final protocol, we will run 9 copies of our segment-lookup protocol segmentLookup. In particular, we will encode each of the 8 types of polynomials $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}, \mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$ for all the circuit branches in a separate table (to be described shortly) and run our segment lookup protocol on each of these 8 tables individually. Let $\{T_i\}_{i\in[8]}$ denote the 8 tables. Each of these T_is has identical sizes and an identical number of segments. Further, even though we run separate instances of segmentLookup on each of them, we require that the *same witness polynomial* $M(X)$ polynomial is used across each copy of segmentLookup(T_i) execution. Note that this can be done naïvely by running the base segmentLookup protocol many times and have the prover sending a *single* round 1 message (which includes the KZG commitment to $[M(\tau)]_1$), which is then subsequently shared across each execution of the protocol.

But we utilize the fact that our segment-lookup protocol inherits linear homomorphism from $\mathtt{cq}$. Specifically, once the prover sends the commitments to the various $F(X)$ polynomials, say $F_1(X)$ and $F_2(X)$ for tables $T_1(X)$ and $T_2(X)$, the verifier samples a random challenge γ, the prover and verifier can run the segment-lookup protocol on $F_1(X) + \gamma F_2(X)$ for the table $T_1(X) + \gamma T_2(X)$. With this observation, we only need to run one copy of the segmentLookup protocol internally.

Polynomials for Segment-Lookup. We are finally ready to describe how we encode the constraints for each circuit branch $\{\mathsf{C}_i\}_{i\in[n]}$ in the 8 tables. We define the following 8 *table* polynomials of degree $ns-1$ in their value representation form - for $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}, \mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$,

$$\forall i \in [0, n-1], j \in [s],\ T_Y(\omega^{is+j}) := Y^{(i)}(\eta^j)$$

where the set $\{0, \eta, \eta^2, \cdots, \eta^{(s-1)}\} := \{0, \omega^n, \omega^{2n}, \cdots, \omega^{(s-1)n}\}$ are the s-th roots of unity. We use $\mathbb{W}$ and $\mathbb{V}$ as defined in Section 3 to be the set of ns and ks roots of unity respectively. If we view T_Y as consisting of n *segments* each of size s, the above equation indicates that the i-th segment consists of $\{Y^{(i)}_j\}_{j\in[s]}$. Similarly, we define the following 8 $ks-1$-degree polynomials, for $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}, \mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$,

$$\forall i \in [0, k-1], j \in [s],\ F_Y(\nu^{is+j}) := Y^{(\xi(i))}(\eta^j)$$

For each $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}, \mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$, we run instances of segmentLookup as described above. For clarity of exposition, we abuse notation and denote this as the P and V executing segmentLookup$^{\otimes \ell}(\{T_i,\}_{i\in[8]})$ (shortening from segmentLookup$^{\otimes \ell}(\{\mathsf{com}_i, T_i, F_i\}_{i\in[8]}, \mathsf{srs}, \mathbb{V})$), where the verifier output is $\{[F_i(\tau)]_1\}_i$ if the proof accepts.

In Section C.1, we describe how the F_Y polynomials, can be easily *post-processed* to obtain a verifier pre-processing identical to that in $\mathcal{P}\mathtt{lon}\mathcal{K}$ for the activated sub-circuit $\widetilde{C}$.

4.2 $\mathcal{S}\mathtt{ublon}\mathcal{K}$ for Layered Branching Circuit

We give a full description of our $\mathcal{S}\mathtt{ublon}\mathcal{K}$ protocol and prove the following theorem in Appendix C.2.

Theorem 2. *Given a $(s, \bar{k}, n, \{\mathsf{C}_i\}_{i=1}^n)$ layered branching circuit C, there exists a pre-processing SNARK in the Algebraic Group Model for $\mathscr{C}$ induced by C with a prover cost of $O(ks \cdot (\log(ks) + \log(n)))$ $\mathbb{G}_1$- and $\mathbb{F}$-operations and a verifier cost and proof size of $O(1)$, where $k \le \bar{k}$ is the effective layer.*

A detailed analysis of the prover cost of our $\mathcal{S}\mathtt{ublon}\mathcal{K}$ protocol can be found in Appendix C.2.

5 IMPLEMENTATION AND EVALUATION

We implement the $\mathcal{S}\mathtt{ublon}\mathcal{K}$ proof system in Rust and are working towards an open-source release. The implementation relies on the BLS12-377 pairing-based curve as implemented in arkworks [6]. Moreover, we rely on the implementation of $\mathcal{P}\mathtt{lon}\mathcal{K}$ available at [3].

All experiments are run on a Macbook Pro with M1 Pro 3.2 Ghz chip and 32 GB RAM. We also report EVM gas costs[12] for publishing and verifying signatures on-chain.

5.1 Implementing and Evaluating $\mathcal{S}\mathtt{ublon}\mathcal{K}$ for Rollups

We demonstrate the improvement of $\mathcal{S}\mathtt{ublon}\mathcal{K}$ on rollup applications.

DEX Rollups. Consider a typical decentralized exchange (DEX) smart contract (e.g. Loopring [1]) which allows users to create one of several types of transactions: deposits, spot trades, transfers, withdrawals, etc. The logic within each of these transaction types is encoded as a circuit, typically ranging from 30K to 60K arithmetic gates – we round up the circuit size to the nearest power of 2 when modeling as a segment, so we use 2^{16} gates[13]. A single instance of a rollup transaction that is submitted to a layer 1 blockchain (e.g. Ethereum) can batch together hundreds of such transactions, along with a single proof attesting to the validity of the next state transition (i.e., the final state of the DEX contract is attained by correctly evaluating the hundreds of transactions starting from the initial state that is recorded on the layer 1 chain). Rollups naturally map to our layered branching circuit model, where at each step, the prover executes one of several transaction types (segments).

Additional Rollup Applications In addition to application-specific rollups, as discussed above with DEX, we experiment with generic rollup solutions which allow for sequentially composing multiple different applications. For each of the following applications (borrowed from [16]), we compose circuits for that application with other similar-sized mock circuits.

[12]Our calculation uses the pre-compiled gas costs for the BLS12-381 curve as defined in EIP-2537 [2]: $\mathbb{G}_1$ additions cost 600, $\mathbb{G}_1$ multiplications cost 12000, $\mathbb{G}_2$ additions cost 800, $\mathbb{G}_2$ multiplications cost 45000, and n pairings cost $115000 + n \cdot 23000$.

[13]This is not a technical limitation, as we could further decompose a segment into a set of smaller segments, each of size a power of two. We have not implemented this idea.

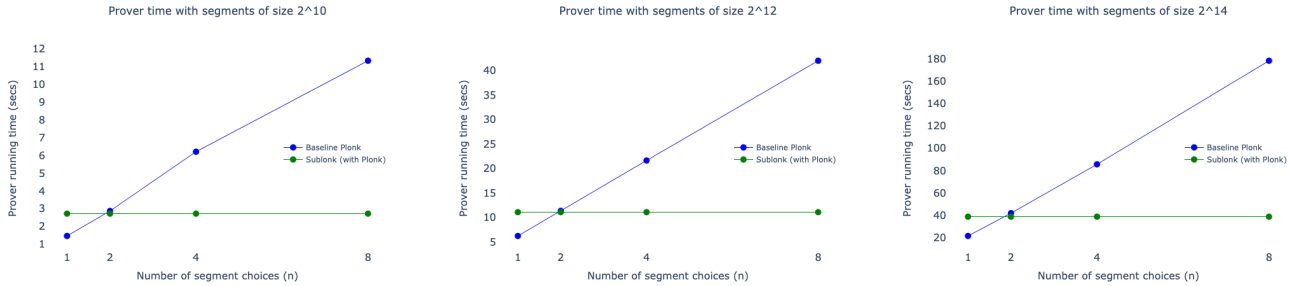

Figure 2: Prover runtime with $k = 128$ segments, with segment choices $n \in \{1, 2, 4, 8\}$, and segments of size 2^{10}, 2^{12}, and 2^{14}

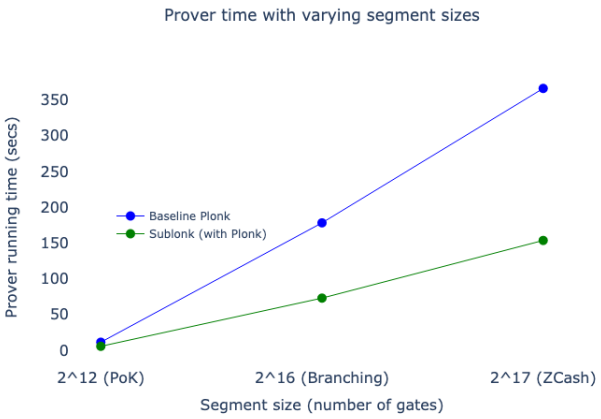

Figure 3: Prover runtime for varying segment sizes

- Proof of knowledge of exponent: each circuit segment encodes the statement that the witness, which is the scalar x, maps to the group element g^x, which is part of the input. We use the BLS12-377 curve (group $\mathbb{G}_1$). The segment has size 2^{12} gates.
- ZCash: each circuit segment encodes the proof circuit for a ZCash [41] (Sapling) transaction. Here, each segment has size 2^{17} gates.
- Nested conditional branching: we consider a generic program containing nested conditional statements, producing a control flow graph comprising a number of possible segments that can be exercised by a particular input – for instance, we report performance for an artificial program with 256 total segments, each of size 2^{16} gates, where an active path only exercises 16 segments.

Note that unlike [16], we do not use custom gates in our circuit encoding, so it results in larger circuits (comparable to the R1CS encoding).

Prover Time. In our first experiment, we encode the DEX rollup computation as a sequence of $k = 128$ steps, where each step has a conditional choice between $n \in \{1, 2, 4, 8\}$ segments, each of size 2^{16} gates; note that n is typically between 4 and 8 for a typical DEX rollup (see [1]), however we also demonstrate SublonK on smaller values of n to better illustrate its performance tradeoffs.

Figure 2 reports the prover time for both SublonK and the baseline PlonK systems. The baseline refers to the invocation of PlonK using the conventional encoding where the circuit representation of all n transaction types are "stitched" together for each of the 128 steps; i.e., the circuit size is $128 \times n \times 2^{16}$ gates. SublonK with PlonK refers to our proof system, which includes our segment-lookup protocol followed by an invocation of PlonK. The overhead incurred by the segment-lookup protocol can be computed by subtracting the SublonK prover time from the baseline prover time for $n = 1$. Not surprisingly, as we increase n, we find larger speedups from SublonK, with over 4.8x speedup for $n = 8$ and segment size 2^{16}. In addition to improvements in the prover time, we also believe that SublonK enables better scaling as it reduces the degree of the polynomials that are provided to the PlonK sub-procedure, thus reducing the memory needs (and likely to execute on smaller machines).

More generally, programs with conditional branches observe a large gap between the total number of segments and the number of executed segments, which makes SublonK's improvement even more significant. As an example, consider a control flow graph of a program with 256 possible segments, each of size 2^{16}. Now consider the an execution, where the active path only exercised 16 segments. The baseline PlonK prover requires roughly 175 seconds to produce a proof, whereas SublonK requires 20.04 seconds; that marks an improvement in prover time of 8.7×.

Finally, we report prover times for the additional rollup applications. In Figure 3, we vary the segment size while fixing $k = 64$ and $n = 4$ – i.e., for each application, the rollup consists of mock circuits of size equal to the segment size, where the rollup prover can choose between 4 possible statements to prove at each of the 64 steps.

Proof Size. Our proof consists of 42 $\mathbb{G}_1$ elements and 12 scalar $\mathbb{F}$ elements, of which 9 $\mathbb{G}_1$ and 6 $\mathbb{F}$ elements arise from the proof generated by the PlonK subprocedure. Concretely, this makes our proof 2.4 KB in size. While the proof is substantially larger compared to the PlonK baseline, which is 624 bytes, we find this to be a valuable performance tradeoff when considering the reductions in prover time.

Verification Cost. The primary operations of the verifier include 23 pairings, 26 $\mathbb{G}_1$ operations, and 1 $\mathbb{G}_2$ operation – if one wishes to strictly enforce that the verifier must be constant time, then one could have the prover compute the opening to the vanishing polynomials, but we find the distinction to be minor in practice due to the efficiency of computing $\log(ns)$ field multiplications. A $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ proof requires 716.6K EVM gas units to verify on-chain and 50 ms to verify on a Macbook Pro laptop, where the computation is dominated by the cost to compute pairings. In comparison, verifying a $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof requires 2 pairings and 18 $\mathbb{G}_1$ multiplications, and costs 377K EVM gas units to verify on-chain.

ACKNOWLEDGMENTS

The second and the fourth authors are supported in part by DARPA under Agreement No. HR00112020026, AFOSR Award FA9550-19-1-0200, NSF CNS Award 1936826, and research grants by the Sloan Foundation and Visa Inc. The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

REFERENCES

[1] 2018. Loopring: A Decentralized Token Exchange Protocol. https://loopring.org/resources/en_whitepaper.pdf
[2] 2020. EIP-2537: Precompile for BLS12-381 curve operations. https://eips.ethereum.org/EIPS/eip-2537
[3] 2022. Jellyfish: A Rust Implementation of the PLONK ZKP System and Extensions. https://github.com/EspressoSystems/jellyfish
[4] 2022. Plonky2: Fast Recursive Arguments with PLONK and FRI. https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf
[5] 2023. Sangria: A Folding Scheme for PLONK. https://geometry.xyz/notebook/sangria-a-folding-scheme-for-plonk
[6] arkworks contributors. 2022. arkworks zkSNARK ecosystem. https://arkworks.rs.
[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *ICALP 2018 (LIPIcs, Vol. 107)*, Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella (Eds.). Schloss Dagstuhl, 14:1–14:17. https://doi.org/10.4230/LIPIcs.ICALP.2018.14
[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046. https://eprint.iacr.org/2018/046.
[9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 459–474. https://doi.org/10.1109/SP.2014.36
[10] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT 2019, Part I (LNCS, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, 103–128. https://doi.org/10.1007/978-3-030-17653-2_4
[11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The Hunting of the SNARK. *J. Cryptol.* 30, 4 (2017), 989–1066. https://doi.org/10.1007/s00145-016-9241-9
[12] Dan Boneh and Xavier Boyen. 2004. Short Signatures Without Random Oracles. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, 56–73. https://doi.org/10.1007/978-3-540-24676-3_4
[13] Benedikt Bünz and Binyi Chen. 2023. ProtoStar: Generic Efficient Accumulation/Folding for Special Sound Protocols. Cryptology ePrint Archive, Paper 2023/620. https://eprint.iacr.org/2023/620
[14] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. 2023. Lookup Arguments: Improvements, Extensions and Applications to Zero-Knowledge Decision Trees. Cryptology ePrint Archive, Paper 2023/1518. https://eprint.iacr.org/2023/1518 https://eprint.iacr.org/2023/1518.
[15] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2075–2092. https://doi.org/10.1145/3319535.3339820
[16] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. 2023. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. In *EUROCRYPT 2023, Part II (LNCS)*. Springer, Heidelberg, 499–530. https://doi.org/10.1007/978-3-031-30617-4_17
[17] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *EUROCRYPT 2020, Part I (LNCS, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26
[18] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile Verifiable Computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 253–270. https://doi.org/10.1109/SP.2015.23
[19] Zijing Di, Lucas Xia, Wilson Nguyen, and Nirvan Tyagi. 2023. MUXProofs: Succinct Arguments for Machine Computation from Tuple Lookups. Cryptology ePrint Archive, Paper 2023/974. https://eprint.iacr.org/2023/974 https://eprint.iacr.org/2023/974.
[20] Liam Eagen, Dario Fiore, and Ariel Gabizon. 2022. cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Report 2022/1763. https://eprint.iacr.org/2022/1763.
[21] Liam Eagen, Dario Fiore, and Ariel Gabizon. 2022. cq: Cached quotients for fast lookups. *IACR Cryptol. ePrint Arch.*, 1763. https://eprint.iacr.org/2022/1763
[22] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. 2018. The Algebraic Group Model and its Applications. In *CRYPTO 2018, Part II (LNCS, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 33–62. https://doi.org/10.1007/978-3-319-96881-0_2
[23] Ariel Gabizon and Dmitry Khovratovich. 2022. flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. Cryptology ePrint Archive, Report 2022/1447. https://eprint.iacr.org/2022/1447.
[24] Ariel Gabizon and Zachary J. Williamson. 2020. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315. https://eprint.iacr.org/2020/315.
[25] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. https://eprint.iacr.org/2019/953.
[26] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT 2013 (LNCS, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Heidelberg, 626–645. https://doi.org/10.1007/978-3-642-38348-9_37
[27] genSTARK. 2020. https://github.com/GuildOfWeavers/genSTARK.
[28] Essam Ghadafi and Jens Groth. 2017. Towards a Classification of Non-interactive Computational Assumptions in Cyclic Groups. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10625)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, 66–96. https://doi.org/10.1007/978-3-319-70697-9_3
[29] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. 2022. Stacking Sigmas: A Framework to Compose Σ-Protocols for Disjunctions. In *EUROCRYPT 2022, Part II (LNCS, Vol. 13276)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Heidelberg, 458–487. https://doi.org/10.1007/978-3-031-07085-3_16
[30] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. 2023. Speed-Stacking: Fast Sublinear Zero-Knowledge Proofs for Disjunctions. In *EUROCRYPT 2023, Part II (LNCS)*. Springer, Heidelberg, 347–378. https://doi.org/10.1007/978-3-031-30617-4_12
[31] Lior Goldberg, Shahar Papini, and Michael Riabzev. 2021. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063. https://eprint.iacr.org/2021/1063.
[32] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. Delegating computation: interactive proofs for muggles. In *40th ACM STOC*, Richard E. Ladner and Cynthia Dwork (Eds.). ACM Press, 113–122. https://doi.org/10.1145/1374376.1374396
[33] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. 2023. Efficient Proofs of Software Exploitability for Real-world Processors. *Proc. Priv. Enhancing Technol.* 2023, 1 (2023), 627–640. https://doi.org/10.56553/popets-2023-0036
[34] Jens Groth. 2010. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT 2010 (LNCS, Vol. 6477)*, Masayuki Abe (Ed.). Springer, Heidelberg, 321–340. https://doi.org/10.1007/978-3-642-17373-8_19
[35] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016, Part II (LNCS, Vol. 9666)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11
[36] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. 2018. Updatable and Universal Common Reference Strings with Applications to

zk-SNARKs. In *CRYPTO 2018, Part III (LNCS, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 698–728. https://doi.org/10.1007/978-3-319-96878-0_24
[37] Ulrich Haböck. 2022. Multivariate lookups based on logarithmic derivatives. *IACR Cryptol. ePrint Arch.*, 1530. https://eprint.iacr.org/2022/1530
[38] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 2055–2074. https://doi.org/10.1145/3372297.3417283
[39] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *EUROCRYPT 2020, Part III (LNCS, Vol. 12107)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 569–598. https://doi.org/10.1007/978-3-030-45727-3_19
[40] hodor. 2021. https://github.com/matter-labs/hodor.
[41] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2016. Zcash protocol specification. *GitHub: San Francisco, CA, USA* 4 (2016), 220.
[42] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *USENIX Security 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1353–1370.
[43] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT 2010 (LNCS, Vol. 6477)*, Masayuki Abe (Ed.). Springer, Heidelberg, 177–194. https://doi.org/10.1007/978-3-642-17373-8_11
[44] Assimakis A. Kattis, Konstantin Panarin, and Alexander Vlasov. 2022. RedShift: Transparent SNARKs from List Polynomial Commitments. In *ACM CCS 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, 1725–1737. https://doi.org/10.1145/3548606.3560657
[45] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. 2020. MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs. In *USENIX Security 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2129–2146.
[46] Abhiram Kothapalli and Srinath Setty. 2022. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758. https://eprint.iacr.org/2022/1758.
[47] Abhiram Kothapalli and Srinath Setty. 2023. HyperNova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573. https://eprint.iacr.org/2023/573
[48] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *CRYPTO 2022, Part IV (LNCS, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Heidelberg, 359–388. https://doi.org/10.1007/978-3-031-15985-5_13
[49] libSTARK. 2018. https://github.com/elibensasson/libSTARK.
[50] Helger Lipmaa. 2016. Prover-Efficient Commit-and-Prove Zero-Knowledge SNARKs. In *AFRICACRYPT 16 (LNCS, Vol. 9646)*, David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.). Springer, Heidelberg, 185–206. https://doi.org/10.1007/978-3-319-31517-1_10
[51] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2111–2128. https://doi.org/10.1145/3319535.3339817
[52] Masip-Ardevol, Héctor, Guzmán-Albiol, Marc, Baylina-Melé, Jordi, and Muñoz-Tapia, Jose Luis. 2023. eSTARK: Extending STARKs with Arguments. Cryptology ePrint Archive, Paper 2023/474. https://eprint.iacr.org/2023/474
[53] Silvio Micali. 1994. CS Proofs (Extended Abstracts). In *35th FOCS*. IEEE Computer Society Press, 436–453. https://doi.org/10.1109/SFCS.1994.365746
[54] Assa Naveh and Eran Tromer. 2016. PhotoProof: Cryptographic Image Authentication for Any Set of Permissible Transformations. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 255–271. https://doi.org/10.1109/SP.2016.23
[55] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 238–252. https://doi.org/10.1109/SP.2013.47
[56] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. 2022. PlonKup: Reconciling PlonK with plookup. Cryptology ePrint Archive, Report 2022/086. https://eprint.iacr.org/2022/086.
[57] Jim Posen and Assimakis A. Kattis. 2022. Caulk+: Table-independent lookup arguments. Cryptology ePrint Archive, Report 2022/957. https://eprint.iacr.org/2022/957.
[58] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. 2022. ZEBRA: Anonymous Credentials with Practical On-chain Verification and Applications to KYC in DeFi. Cryptology ePrint Archive, Paper 2022/1286. https://eprint.iacr.org/2022/1286 https://eprint.iacr.org/2022/1286.
[59] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. 2023. Unlocking the lookup singularity with Lasso. *IACR Cryptol. ePrint Arch.* (2023), 1216.
[60] Meilof Veeningen. 2017. Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation. In *AFRICACRYPT 17 (LNCS, Vol. 10239)*, Marc Joye and Abderrahmane Nitaj (Eds.). Springer, Heidelberg, 21–39.
[61] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society.
[62] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. 2022. Caulk: Lookup Arguments in Sublinear Time. In *ACM CCS 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, 3121–3134. https://doi.org/10.1145/3548606.3560646
[63] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. 2022. Caulk: Lookup Arguments in Sublinear Time. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 3121–3134. https://doi.org/10.1145/3548606.3560646
[64] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. 2022. Baloo: Nearly Optimal Lookup Arguments. Cryptology ePrint Archive, Report 2022/1565. https://eprint.iacr.org/2022/1565.
[65] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero Knowledge Proofs for Decision Tree Predictions and Accuracy. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 2039–2053. https://doi.org/10.1145/3372297.3417278
[66] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. A Zero-Knowledge Version of vSQL. *IACR Cryptol. ePrint Arch.* (2017), 1146. http://eprint.iacr.org/2017/1146
[67] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster Verifiable RAM with Program-Independent Preprocessing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 908–925. https://doi.org/10.1109/SP.2018.00013
[68] ZkRollups. 2021. An incomplete guide to rollups. https://vitalik.ca/general/2021/01/ 05/rollup.html.

A ADDITIONAL PRELIMINARIES

A.1 Algebraic Group Model

We use the same terminology as prior works [20, 22, 25]. An *algebraic adversary* $\mathcal{A}$ in an SRS-based protocol is a $\text{poly}(\lambda)$-time algorithm which satisfies the following: For each $i \in \{1, 2\}$, whenever $\mathcal{A}$ outputs a group element $G \in \mathbb{G}_i$, it also outputs a vector v over $\mathbb{F}$ such that $G = \langle v, \text{srs}_i \rangle$. srs is said to have degree d if all elements of srs_i are of the form $[F(\tau)]_i$ for $F \in \mathbb{F}_{<d}[X]$ and uniform $\tau \in \mathbb{F}$. In the following, it is assumed that a degree d SRS is used. Let $F_{i,j}$ denote the corresponding polynomial for the jth element of srs_i.

We require the following d-DLOG assumption to ensure that the srs in our protocol hides the random $\tau \in \mathbb{F}$ from $\mathcal{A}$.

Definition 3 (d-DLOG Assumption [20, 22]). *Fix an integer d. The d-DLOG assumption for $(\mathbb{G}_1, \mathbb{G}_2)$ ensures that given $[1]_1.[\tau]_1, \cdots, [\tau^d]_1, [1]_2, [\tau]_2, \cdots, [\tau^d]_2$ for a uniformly random $\tau \in \mathbb{F}$, the probability of $\mathcal{A}$ outputting τ is $\text{negl}(\lambda)$.*

Furthermore, we require the following lemma from [20] which ensures that if the "*real*" pairing checks in our protocol are guaranteed to pass, then so would the "*ideal*" pairing check, where essentially the algebraic adversary $\mathcal{A}$ also gives a field vector corresponding to the group elements in the pairing check (as described above).

Lemma 1 ([20]). *Assume d-DLOG assumption for $(\mathbb{G}_1, \mathbb{G}_2)$. Given an algebraic adversary $\mathcal{A}$, and the d-degree* srs *from our protocol, the probability of a real pairing check passing is larger than the corresponding ideal check, by at most an additive factor of* $\text{negl}(\lambda)$. *The real and ideal pairing checks are described below.*
Real Pairing Check: For $\mathbb{F}$-vectors a, b, whose $\mathbb{G}_1, \mathbb{G}_2$ encodings are given by $\mathcal{A}$ during the protocol execution, the real check is of the

form: $(a \cdot T_1) \cdot (T_2 \cdot b) = 0$*, for some matrices* T_1, T_2 *over* $\mathbb{F}$*. Such a check is done efficiently given the encoded group elements using the pairing function* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$*.*
Ideal Pairing Check: Since $\mathcal{A}$ *is algebraic, for each group encoding* $[a_j]_i$ *corresponding to* a_j *in the vector* a*, it also outputs a vector* v *such that* $a_j = \sum v_\ell F_{i,\ell}(\tau) = R_{i,j}(\tau)$*, for* $R_{i,j}(X) := \sum v_\ell F_{i,\ell}(X)$*. For* $i \in \{1, 2\}$*, we denote the vector of polynomials here by* $R_i = (R_{i,j})_j$*. The ideal check is of the form:* $(R_1 \cdot T_1) \cdot (T_2 \cdot R_2) \equiv 0$*.*

A.2 Batched KZG Commitments

Definition 4 (Batched KZG Commitment [25]). *A* d*-polynomial commitment scheme consists of*

- $\mathsf{KZG.Gen}(d)$*: randomized algorithm that outputs* srs_{kzg}*.*
- $\mathsf{KZG.Commit}(P, \mathsf{srs}_{kzg})$ *: given a polynomial* $P \in \mathbb{F}_{<d}[X]$*, outputs a commitment* com *to* P*.*
- *A public coin protocol between a prover and a verifier. The prover gets* $P_1, \cdots, P_t \in \mathbb{F}_{<d}[X]$*. Both the parties get the inputs* $t = \mathsf{poly}(\lambda)$*,* $z_1 \cdots, z_t \in \mathbb{F}$*,* $\mathsf{com}_1, \cdots, \mathsf{com}_t$*, the alleged commitments to* $P_1, \cdots, P_t$*, and* $s_1, \cdots, s_n \in \mathbb{F}$*, the alleged correct openings* $P_1(z_1), \cdots, P_t(z_t)$*. At the end of the protocol the verifier outputs accept or reject.*

such that the following properties hold

- ***Completeness:*** *Fix integer* $t, z_1, \cdots, z_t \in \mathbb{F}, P_1, \cdots, P_t \in \mathbb{F}_{<d}[X]$*. Suppose that for each* $i \in [t]$*,* $\mathsf{com}_i = \mathsf{KZG.Commit}(P_i, \mathsf{srs}_{kzg})$*. Then if the prover and verifier honestly run the protocol with inputs* $t, \{\mathsf{com}_i, z_i, s_i = P_i(z_i)\}_{i\in[t]}$*, the verifier accepts w.p. 1.*
- ***Knowledge Soundness in the algebraic group model:*** *The probability of any efficient algebraic adversary* $\mathcal{A}$ *winning the following game is* $\mathsf{negl}(\lambda)$*.*
  - *Given* srs_{kzg}*,* $\mathcal{A}$ *outputs* $t, \mathsf{com}_1, \cdots, \mathsf{com}_t$ *along with polynomials* $P_1, \cdots, P_t \in \mathbb{F}_{<d}[X]$*, corresponding to the commitments.*
  - $\mathcal{A}$ *outputs* $z_1, \cdots, z_t, s_1, \cdots, s_t \in \mathbb{F}$*.*
  - $\mathcal{A}$ *(as a prover) and the verifier run the commitment protocol with inputs* $\mathsf{com}_1, \cdots, \mathsf{com}_t, z_1, \cdots, z_t, s_1, \cdots, s_t$*.*
  - $\mathcal{A}$ *wins if and only if the verifier accepts and for some* $i \in [t]$*,* $s_i \neq P_i(z_i)$*.*

A.3 Pre-processing SNARKs in the Algebraic Group Model

In this section, we provide a formal definition of the pre-processing SNARKs in the algebraic group model, as defined in [25].

Definition 5 (SNARKs in the Algebraic Group Model [25]). *Let* $\mathcal{R}$ *be a relation generator that given a security parameter* λ *in unary returns a polynomial time decidable binary relation* R*. For pairs* $(\phi, w) \in R$ *we call* ϕ *the statement and* w *the witness. We define* $\mathcal{R}_\lambda$ *to be the set of possible relations* R *that the relation generator may output given* 1^λ*. We will in the following for notational simplicity assume* λ *can be deduced from the description of* R*. The relation generator may also output some side information, an auxiliary input* z*, which will be given to the adversary. An efficient prover publicly verifiable non-interactive argument for* $\mathcal{R}$ *is a quadruple of probabilistic polynomial algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ *defined as:*

- $\mathsf{srs} \leftarrow \mathsf{Setup}(1^\lambda, R)$*: The setup takes the security parameter* λ *and the relation* R *as input and produces a structured reference string* srs*.*
- $\pi \leftarrow \mathsf{Prove}(\mathsf{srs}, R, \phi, w)$*: The prover algorithm takes as input* srs *and* $(\phi, w) \in R$ *and returns an argument* π*.*
- $0/1 \leftarrow \mathsf{Ver}(\mathsf{srs}, R, \phi, \pi)$*: The verification algorithm takes as input* srs*, a statement* ϕ *and an argument* π *and returns 0 (reject) or 1 (accept).*

We say that $\Sigma = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver})$ *is a SNARK in the algebraic group model if it satisifes the following properties.*

- ***Completeness:*** *Given any true statement, an honest prover should be able to convince an honest verifier. For all* $\lambda \in \mathbb{N}$*,* $R \in \mathcal{R}_\lambda$*,* $(\phi, w) \in R$

$$\Pr\Big[\mathsf{srs} \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathsf{Prove}(\mathsf{srs}, R, \phi, w) : \mathsf{Ver}(\mathsf{srs}, R, \phi, w) = 1\Big] \geq 1 - \mathsf{negl}(\lambda).$$

- ***Knowledge soundness in the algebraic group model:*** *The probability of an efficient algebraic adversary winning in the following game should be* $\mathsf{negl}(\lambda)$*.*
  - *Given the* srs *and* R*,* $\mathcal{A}$ *outputs* (ϕ, π) *along with the corresponding witness* w*.*
  - $\mathcal{A}$ *wins if and only if the verifier accepts and* $(\pi, w) \notin \mathcal{R}$*.*
- ***Succinctness.*** *A non-interactive argument of knowledge where the verifier runs in polynomial time in* $\lambda + |\phi| + \log(|R|)$ *and the proof size is polynomial in* $\lambda + \log(|R|)$ *is called a pre-processing SNARK. If we also restrict the* srs *to be polynomial in* $\lambda + \log(|R|)$ *we say that the non-interactive argument is a fully succinct SNARK.*

A.4 Preliminary Lemmas

We require several properties of polynomials over fields for our protocol, which are described below. We require the following lemma on the uniqueness of fractional representations from [20].

Lemma 2. *[20, Lemma 4] Let* $\mathbb{F}$ *be an arbitrary field and* $m_1, m_2 : \mathbb{F} \to \mathbb{F}$ *be any functions. Then* $\sum_{z\in\mathbb{F}} \frac{m_1(z)}{X-z} = \sum_{z\in\mathbb{F}} \frac{m_2(z)}{X-z}$ *in the rational field* $\mathbb{F}(X)$*, if and only if,* $m_1(z) = m_2(z)$ *for every* $z \in \mathbb{F}$*.*

Sumcheck Lemma. We require the following sumcheck lemma from [10, 20].

Lemma 3. *Let* $H \subset \mathbb{F}$ *be a multiplicative subgroup of size* t*. For* $F \in \mathbb{F}_{<t}[X]$*, we have*

$$\sum_{a\in H} F(a) = t \cdot F(0).$$

We require the following pre-processing lemma that combines [20, Lemma 3.1] and [20, Theorem 1].

Lemma 4. *Let* $G \in \mathbb{F}_{<ns}[X]$ *and a subgroup* $\mathbb{W} \subset \mathbb{F}$ *of size* ns*. Further, suppose the* $\mathbb{G}_1$ *elements* $\{[\tau^i]_1\}_{i\in[0,ns-1]}$ *are given. Then, the following two computations are possible:*

(1) For each $i \in [ns]$*, it is possible to compute the elements* $q_i := [Q_i(\tau)]_1$*, where* $Q_i(X) \in \mathbb{F}[X]$ *is such that*

$$\psi_i^W(X) \cdot G(X) = g_i \cdot \psi_i^W(X) + Z_{\mathbb{W}}(X) \cdot Q_i(X),$$

for $g_i = G(\omega^i)$*, in* $O(ns \log(ns))$ $\mathbb{G}_1$ *operations.*

(2) Furthermore, there is an algorithm, that takes as input, the q_i*'s and* $[\psi_i^{\mathbb{W}}(\tau)]_1$ *for each* $i \in [ns]$ *from step 1 above, and a* ks*-sparse polynomial* $H(X) \in \mathbb{F}_{<ns}[X]$ *given in the sparse representation, and does the following. It takes* $O(ks)$ $\mathbb{F}$*-operations*

and ks $\mathbb{G}_1$-operations to compute $[Q_A(\tau)]_1$ and $[R_A(\tau)]_1$, where $Q_A(X), R_A(X) \in \mathbb{F}_{<ns}[X]$ are such that

$$H(X)G(X) = Q_H(X)Z_{\mathbb{W}}(X) + R_H(X).$$

B PROTOCOL FOR SEGMENT LOOKUP

B.1 Building Blocks

We require two main building blocks for our segment-lookup protocol, which we state as two lemmas below. The first lemma gives a protocol that helps check if some polynomial evaluations are all n-th roots of unity, which can be instantiated using Caulk [62]. The second lemma is a modification of the log-derivative lemma from [20, 37].

Caulk Multi-Unity Sub-Protocol. We state the instantiation of Multi-unity sub-protocol from Caulk [62, Theorem 4] in Lemma 5 below and give an informal description of the protocol subsequently.

Lemma 5. *If the qSDH, qDHE, and qSFrac assumptions [12, 28] hold, then there exists a knowledge-sound argument for $\mathcal{R}_{\mathsf{unity}} := \{(\mathsf{srs}, [D]_1, \{\nu^{is} : i \in [0, k-1]\}, \{\mu^i : i \in [0, n-1]\}) : [D]_1 = [D(\tau)]$ for $D(x)$ s.t. $\forall i \in [0, k-1]$, $D(\nu^{is}) = \mu^j$ for some $j \in [0, n-1]\}$, where μ is some n-th root of unity, under the algebraic group model and random oracle model. Moreover, the prover uses $O(ks \log n)$ $\mathbb{G}_1$- and $\mathbb{F}$-operations, while the proof size and verifier cost are constant.*

We now give an overview of the Multi-unity sub-protocol from Caulk. The aim of the protocol is to check if the evaluations of some polynomial are all roots of unity, i.e., prove that for each $i \in [0, k-1], D(\nu^{is}) \in \{\mu^0, \ldots, \mu^{n-1}\}$, where $\mu = \omega^s$ and $\{\mu^0, \ldots, \mu^{n-1}\}$ are the n^{th} roots of unity. To do this, the prover defines a vector $\vec{u}_0 = (D(1), D(\nu^s), \ldots, D(\nu^{(k-1)s}))$, and iteratively defines $\vec{u}_j = \vec{u}_{j-1} \circ \vec{u}_{j-1}$, for all $j \in [\log n]$. This means the following must be proved:

(1) $\vec{u}_0$ consists of $(D(1), D(\nu^s), \ldots, D(\nu^{(k-1)s}))$.
(2) $\vec{u}_j = \vec{u}_{j-1} \circ \vec{u}_{j-1}$ for all $j \in [\log n - 1]$.
(3) $\vec{u}_{\log n - 1} \circ \vec{u}_{\log n - 1} = \vec{1}$.

Proving these three statements would imply that the initial statement is proved to hold. To prove steps 1-3 in terms of the polynomial encodings, set $U_0(X) = D(X)$, the polynomial with evaluations $\vec{u}_0$ on $\mathbb{K}$, $U_{\log n}(X) = \mathsf{id}(X)$, where id is the polynomial evaluating to 1 at all points of $\mathbb{V}$, and $U_j(X)$'s to encode the intermediate vectors $\vec{u}_j$'s. Then, it needs to be proved that the following equations hold.

$$U_0(X)U_0(X) - U_1(X) \equiv Z_{\mathbb{V}}(X)Q_{U,1}(X)$$
$$U_1(X)U_1(X) - U_2(X) \equiv Z_{\mathbb{V}}(X)Q_{U,2}(X)$$
$$\vdots$$
$$U_{\log n-1}(X)U_{\log n-1}(X) - \mathsf{id}(X) \equiv Z_{\mathbb{V}}(X)Q_{U,\log n}(X)$$

All the above checks are aggregated into one equation, using lagrange polynomials $\{\Delta_i\}_{i\in[\log n]}$ over roots of unity $\mathbb{S} = \{1, \phi, \ldots, \phi^{\log n-1}\}$ as follows.

$$(U_0^2(X)\Delta_1(Y) + \sum_{j=2}^{\log n} U_{j-1}^2(X)\Delta_j(Y)) - (\sum_{j=1}^{\log n-1} U_j(X) \cdot \Delta_j(Y) + \mathsf{id}(X)\Delta_{\log n}(Y)) = Z_{\mathbb{V}}(X)Q_2(X, Y)$$

for some polynomial $Q_2(X, Y)$. It needs to be shown that the above equation holds at some random point (ζ, ξ) using bivariate KZG. We refer the reader to Appendix D for a formal description of our exact instantiation of the above subprotocol along with its prover cost details.

Log-derivative Method. We use a variant of the log-derivative method from [20, 37] that is tailored for our segment lookup. We state the lemma below, whose proof is similar to the log-derivate lemma from [37].

Lemma 6. *Let $\mathbb{F}$ be a field with characteristic $p > \max\{ks, ns\}$. Given three sequences of field elements $f = (f_i = F(\nu^i))_{i\in[ks]}$, $\ell = (\ell_i = L(\nu^i))_{i\in[ks]}$, and $t = (t_i = T(\omega^i))_{i\in[ns]}$. Consider segments of s-consecutive elements in f and t, resulting in k and n segments of f and t, respectively. We have that for each $i \in [0, k-1]$, the i-th segment of f corresponds to the ℓ_i-th segment of t, i.e., $\forall\delta \in \mathbb{F}$, for each $i \in [ks]$, there exists some $j \in [ns]$ such that $f_i + \delta\ell_i = t_j + \delta\omega^j$, if and only if, for some $m \in \mathbb{F}^{ns}$, the following identity of rational functions holds.*

$$\sum_{i\in[ns]} \frac{m_i}{X + t_i + \delta\omega^i} = \sum_{i\in[ks]} \frac{1}{X + f_i + \delta\ell_i} \tag{1}$$

Proof. Denote by $m_{f,\ell}(z)$, the multiplicity of field element z in the sequence $(f_i + \delta\ell_i)_{i\in[ks]}$. Since $p > \max\{ks, ns\}$, the multiplicities will be non-zero elements in $\mathbb{F}$. Suppose that for each $i \in [ks]$, there exists some $j \in [ns]$ such that for each $\delta \in \mathbb{F}$, it holds that $f_i + \delta\ell_i = t_j + \delta\omega^j$. Then, set $m_j = m_{f,\ell}(t_j + \delta\omega^j)$ for the $j \in [ns]$ corresponding to the i's above, and to be 0 for the remaining j's in $[ns]$. Clearly, for these choice of m_j's, equation 1 holds.
Conversely, suppose equation 1 holds for each $\delta \in \mathbb{F}$. By collecting repeating terms of each of the summands in the field, we get:

$$\sum_{i\in[ns]} \frac{m_i}{X + t_i + \delta\omega^i} = \sum_{z\in\mathbb{F}} \frac{m'(z)}{X + z},$$
$$\sum_{i\in[ks]} \frac{1}{X + f_i + \delta\ell_i} = \sum_{z\in\mathbb{F}} \frac{m_{f,\ell}(z)}{X + z}.$$

where $m'(z)$ is $m_t(z)$ multiplied with the corresponding m_i's. Since, $p > \max\{ks, ns\}$, for each $z \in \{f_i + \delta\ell_i\}$, we have $m_{f,\ell}(z) \neq 0$. By uniqueness of fractional representations (Lemma 2), $m_f(z) = m'(z)$ for each $z \in \{f_i\}$. Hence, for each $z \in \{f_i + \delta\ell_i\}$, there must exist some $j \in [ns]$ such that $z = t_j + \delta\omega^j$. □

Tables with Repeated Elements A crucial feature implicit in the proof of Lemma 6 is that it assumes that the table T consists of unique values. We give details of this and show how to handle repeated table entries below.

An astute reader may have noted that the Lemma 6 only works if within each *segment* the table T consists of unique values, i.e. $\forall i \in [0, n-1] \nexists j, j' \in [s]$ such that $T(\omega^{is+j}) = T(\omega^{is+j'})$. At a high level, this is implicit in the proof of Lemma 6. Particularly, if there is a repeated element in a segment, i.e., 2 elements in the table correspond to the same t_j, then the multiplicity count argument for proving equation 1 in the proof will no longer hold.

Looking ahead, in our application of the segment lookup protocol, we will indeed deal with tables that will contain repeated elements with a segment. We now describe how to generically

start with any table $T(X)$, to (i) convert it into a table $T'(X)$ during the pre-processing step such that each segment consists of unique values - the prover will run the protocol using $F'(X)$ derived from $T'(X)$; and (ii) have the verifier perform a single step to *recover* $F(X)$ from $F'(X)$. The guarantee of our transformation will be that the *witness polynomial* is identical in both cases. Specifically, our segment lookup protocol when run on T' and F' ensures that $\exists$ a function $\xi : [k] \to [N]$ such that $\forall i \in [0,k], j \in [s], F'(v^{is+j}) = T'(v^{\xi(i)s+j})$, and our transformation will ensure that $\forall i \in [0,k], j \in [s], F(v^{is+j}) = T(v^{\xi(i)s+j})$ for the *same* function ξ.

Let $T_{\max} := \max_i |T(\omega^i)|$, and we shall set $\forall i \in [0, n-1], j \in [s], T'(\omega^{is+j}) := T(\omega^{is+j}) + 2 \cdot j \cdot (T_{\max}+1)$. This is done by further defining two polynomials, $E(X) = \sum_{i=0}^{n-1} \sum_{j\in[s]} 2\cdot j\cdot(T_{\max}+1)\psi^{\mathbb{W}}_{is+j}(X)$ and $D(X) = \sum_{i=0}^{k-1} \sum_{j\in[s]} 2 \cdot j \cdot (T_{\max}+1)\psi^{\mathbb{V}}_{is+j}(X)$. We define the following function makeUnique:

makeUnique(T, E): Output $T'(X) := T(X) + E(X)$.

Within the context of our segment lookup protocol, $[T'(\tau)]$ and $[D(\tau)]$ are computed as a part of pre-processing by running makeUnique(T, E), when given $T(X)$ as input - the rest of the pre-processing remains unchanged. The prover sets $F'(X)$ to be $F'(X) = F(X) + D(X)$, and the verifier sets $[F'(\tau)] = [F(\tau)] + [D(\tau)]$ to run segmentLookup$(\mathsf{com}, \mathsf{srs}, T', F', \mathbb{V})$, and verifier setting the "output" of the protocol to be $[F(\tau)]$. For correctness, we rely on the following claims.

Claim 1. *If* $T_{\max} \cdot (2 \cdot s + 1) < |\mathbb{F}|$, *then* $\forall i \in [N] \nexists j, j' \in [s]$ *such that* $T'(\omega^{is+j}) = T'(\omega^{is+j'})$.

Proof Sketch. If not, for any distinct $j, j' \in [s]$ consider $T'(\omega^{is+j}) - T'(\omega^{is+j'}) = T(\omega^{is+j}) - T(\omega^{is+j'}) + 2(T_{\max}+1)(j-j')$. Since by definition, $T(\omega^{is+j}) - T(\omega^{is+j'}) \in [-2T_{\max}, 2T_{\max}]$, we have that $T'(\omega^{is+j}) - T'(\omega^{is+j'}) > 0$ if $T_{\max} \cdot (2 \cdot s + 1) < |\mathbb{F}|$, and thus $T'(\omega^{is+j}) \neq T'(\omega^{is+j'})$. □

Note that we are assuming that $T_{\max} \cdot (2 \cdot s + 1) < |\mathbb{F}|$ to avoid any "wrap-around" in the field. This restriction will not hinder the applications we consider.

Claim 2. $\exists$ *a function* $\xi : [k] \to [N]$ *such that* $\forall i \in [0,k], j \in [s], F'(v^{is+j}) = T'(v^{\xi(i)s+j})$, *then* $\forall i \in [0,k], j \in [s], F(v^{is+j}) = T(v^{\xi(i)s+j})$.

The proof of the above claim follows from the definition of the relevant polynomials.

For the rest of this work, we will assume that we invoke segmentLookup, we are running the above modified version of the segment lookup protocol that deals with repeated elements within a segment. Thus, we ignore this issue henceforth.

B.2 Protocol Description

We now give a formal description of our segment-lookup protocol.

gen(n, k, s, T)**:** Given n, k, s and the polynomial $T(X) \in \mathbb{F}[X]$ of degree $ns-1$, the pre-processing information is computed as follows:

The pre-processing involves computing all the powers of a random τ *(step 1) which is the only step of pre-processing requiring a trusted setup. In the remaining steps, these trusted powers of* τ *are used to generate the commitments of the relevant vanishing polynomials, the Lagrange polynomials, and some quotient polynomials for reducing prover and verifier work. These quotient polynomial computations are key to reducing the online prover cost.*

(1) Choose a random $\tau \in \mathbb{F}$. Let $\max = \max(k, n)$. Compute $\{[\tau^i]_1\}_{i\in[0,\max\cdot s-1]}$ and $\{[\tau^i]_2\}_{i\in[0,\max\cdot s-1]}$.
(2) Compute $[Z_{\mathbb{W}}(\tau)]_2$, $[Z_{\mathbb{V}}(\tau)]_2$ and $[Z_{\mathbb{K}}(\tau)]_2$.
(3) Compute and output $[T(\tau)]_2$.
(4) For $i \in [ns]$, use lemma 4 to compute:
   (a) $q_{i,1} = [Q_{i,1}(\tau)]_1$ and $q_{i,2} = [Q_{i,2}(\tau)]_1$ such that
   $$\psi_i^{\mathbb{W}}(X) \cdot T(X) = t_i \cdot \psi_i^{\mathbb{W}}(X) + Z_{\mathbb{W}}(X) \cdot Q_{i,1}(X)$$
   $$\psi_i^{\mathbb{W}}(X) \cdot X = \omega^i \cdot \psi_i^{\mathbb{W}}(X) + Z_{\mathbb{W}}(X) \cdot Q_{i,2}(X)$$
   (b) $[\psi_i^{\mathbb{W}}(\tau)]_1$.
   (c) $[\frac{\psi_i^{\mathbb{W}}(\tau) - \psi_i^{\mathbb{W}}(0)}{\tau}]_1$.
(5) For $i \in [ks]$, compute the commitments $[\psi_i^{\mathbb{V}}(\tau)]_1$, $[\psi_i^{\mathbb{V}}(\tau\nu)]_1$, for all $i \in [ks]$, corresponding to the set $\mathbb{V}$.
(6) For $i \in [ns]$, use lemma 4 to compute:
   (a) $q_{i,3} = [Q_{i,3}(\tau)]_1$ and $q_{i,4} = [Q_{i,4}(\tau)]_1$ such that:
   $$\psi_i^{\mathbb{W}}(X) \cdot (X^n - 1) = g_i \psi_i^{\mathbb{W}}(X) + Z_{\mathbb{W}}(X) \cdot Q_{i,3}(X)$$
   $$\psi_i^{\mathbb{W}}(X/\omega) \cdot (X^n - 1) = g_i \psi_i^{\mathbb{W}}(X/\omega) + Z_{\mathbb{W}}(X) \cdot Q_{i,4}(X),$$
   where $g_i := (\omega^{in} - 1)$.
   (b) $[\psi_i^{\mathbb{W}}(\tau/\omega)]_1$.

Output srs: $\{[\tau^i]_1, [\tau^i]_2\}_{i\in[0,\max\cdot s-1]}$, $[Z_{\mathbb{W}}(\tau)]_2$,$[Z_{\mathbb{V}}(\tau)]_2$, $[Z_{\mathbb{K}}(\tau)]_2$, $[T(\tau)]_2$, $q_{i,1}$, $q_{i,2}$, $q_{i,3}$, $q_{i,4}$, $[\psi_i^{\mathbb{W}}(\tau)]_1$, $[\psi_i^{\mathbb{W}}(\tau/\omega)]_1$, and $[\frac{\psi_i^{\mathbb{W}}(\tau) - \psi_i^{\mathbb{W}}(0)}{\tau}]_1$ for each $i \in [ns]$, and $[\psi_i^{\mathbb{V}}(\tau)]_1$, $[\psi_i^{\mathbb{V}}(\tau\nu)]_1$ for each $i \in [ks]$.

segmentLookup$(\mathsf{com}, \mathsf{srs}, T, F, \mathbb{V})$**:** This protocol proceeds as follows:

In Round 1*, the prover first commits to the multiplicity vector* m *from eq. 1. Then, the prover computes the vector* ℓ*, such that* ℓ_i *corresponds to the segment of* t *that matches the* i*-th segment of* f*, and generates commitments of two polynomials encoding* ℓ_i*:* L*, which helps in checking that within each of the* i*-th segment, the consecutive elements of* f *correspond to consecutive elements of* t *in the* ℓ_i*-th segment, and* D*, which helps in checking that* L *is well-formed, i.e., the first entries of each segment form a set of* n*-th roots of unity. Note here that all the commitments on* τ *(which the prover does not know) are computed by taking a linear combination of the corresponding Lagrange commitments, all of which are given in the* srs.

Round 1 (Prover → Verifier). The prover does the following:

(1) Computes a polynomial M of degree $ns-1$ as follows: For each $i \in [0, n-1]$, if i^{th} segment in t is executed y times in $F|_{\mathbb{V}}$, then for each $j \in [0, s-1]$, $M(\omega^{is+j}) = y$, and sends $[M(\tau)]_1$ and $[M(\tau/\omega)]_1$ to the verifier.
(2) Computes and sends $[Q_M(\tau)]_1$ using the srs and lemma 4, where $Q_M(X)$ is such that[14]
$$(X^n - 1)(M(X) - M(X/\omega)) = Z_{\mathbb{W}}(X) Q_M(X) \quad (2)$$

[14]Given $q_{i,3}, q_{i,4}, [\psi_i^{\mathbb{W}}(\tau)]_1, [\psi_i^{\mathbb{W}}(\tau/\omega)]_1$, for each $i \in [ns]$, we can slightly modify the step 2 of lemma 4 to compute a linear combination and obtain $[Q_M(\tau)]_1$ and the commitment of the remainder polynomials corresponding to the ks-sparse polynomial $H(X) := (M(X) - M(X/\omega))$.

where $Z_{\mathbb{W}}(X)$ denotes the vanishing polynomial corresponding to the set $\mathbb{W}$.

(3) Computes $L(X)$ of degree $ks-1$ as follows: For each $i \in [0, k-1]$, if the segment executed at the i^{th} step (i.e., the i^{th} segment in $f = F|_{\mathbb{V}}$) is the j^{th} segment in t, then for each $q \in [0, s-1]$, $L(v^{is+q}) = \omega^{js+q}$, and sends $[L(\tau)]_1$ and $[L(\tau v)]_1$ to the verifier.

(4) Computes $Q_L(X)$ such that

$$(X^k - 1)(L(Xv) - \omega L(X)) = Z_{\mathbb{V}}(X)Q_L(X) \tag{3}$$

where $Z_{\mathbb{V}}(X)$ denotes the vanishing polynomial corresponding to the set $\mathbb{V}$, and sends $[Q_L(\tau)]_1$ to the verifier.

(5) Computes a degree $k-1$ polynomial $D(X)$ such that for each $i \in [0, k-1]$, $D(v^{is}) = L(v^{is})$, and sends $[D(\tau)]_1$ to the verifier.

(6) Computes Q_D such that

$$L(X) - D(X) = Z_{\mathbb{K}}(X)Q_D(X) \tag{4}$$

where $Z_{\mathbb{K}}(X)$ denotes the vanishing polynomial corresponding to the set $\mathbb{K}$, and sends $[Q_D(\tau)]_1$ to the verifier.

***In Round 2**, the verifier checks validity of the equation 2 at τ. For soundness, it is important that the eqs 3 and 4 corresponding to L and D in Round 1 are checked at random challenges. For efficiency reasons, we combine these checks with the other checks in rounds 14-15.*

Round 2 (Verifier). The verifier proceeds to check the following:

$$e([M(\tau)]_1 - [M(\tau/\omega)]_1, [\tau^n]_2 - [1]_2) = e([Q_M(\tau)]_1, [Z_{\mathbb{W}}(\tau)]_2)$$

***In Rounds 3-8**, the prover and verifier run the sub-protocol from [62, Figure 5], as described in Section B.1* to check that the first entries of each segment $i \in [0, k-1]$ of f, form the n-th roots of unity $v^0, \ldots, v^{n-1}$, for $v = \omega^s$. Recall, this was needed to guarantee the correctness of L.

Round 3-8 (Prover $\longleftrightarrow$ Verifier). Using the instantiation of Lemma 5, the prover and verifier engage in a protocol to prove that the polynomial L is well-formed, i.e., for each $i \in [0, k-1]$, $D(v^{is}) \in \{\mu^0, \ldots, \mu^{n-1}\}$, where $\mu = \omega^s$ and $\{\mu^0, \ldots, \mu^{n-1}\}$ are the n^{th} roots of unity.

***In Rounds 8-10**, the prover computes the polynomials A and B corresponding to the two summations in equation 1, and sends the needed commitments for the verifier to check the correctness of A at τ, and to perform degree-check of the polynomial that has the least degree amongst A and B at τ. No such degree check is needed if $k = n$.*

Round 9 (Verifier). The verifier sends random $\beta, \delta \in \mathbb{F}$ to the prover.

Round 10 (Prover $\rightarrow$ Verifier). The prover does the following:

(1) The prover computes $A(X)$ of degree $ns-1$ such that for each $i \in [0, ns-1]$,

$$A(\omega^i) = \frac{M(\omega^i)}{\beta + T(\omega^i) + \delta\omega^i}$$

and sends $[A(\tau)]_1$ to the verifier.

(2) The prover computes $[Q_A(\tau)]_1$ using the srs and step 2 of lemma 4, where $Q_A(X)$ is such that[15]

$$A(X)(\beta + T(X) + \delta X) - M(X) = Z_{\mathbb{W}}(X)Q_A(X).$$

(3) The prover computes $B(X)$ of degree $ks-1$ such that for each $i \in [0, ks-1]$,

$$B(v^i) = \frac{1}{\beta + F(v^i) + \delta L(v^i)}$$

and sends $[B(\tau)]_1$ to the verifier.

(4) The prover computes $Q_B(X)$ such that

$$B(X)(\beta + F(X) + \delta L(X)) - 1 = Z_{\mathbb{V}}(X)Q_B(X),$$

and sends $[Q_B(\tau)]_1$ to the verifier.

(5) The prover computes $B_0(X) = \frac{B(X)-B(0)}{X}$, $A_0(X) = \frac{A(X)-A(0)}{X}$, and sends $[A_0(\tau)]_1, [B_0(\tau)]_1$ to the verifier.

(6) **For degree check**, which is needed only for $k \neq n$,
if $n > k$: the prover computes $P_B(X) = B_0(X) \cdot X^{ns-(ks+1)}$ and sends $[P_B(\tau)]_1$ to the verfier;
else if $k > n$: the prover computes $P_A(X) = A_0(X) \cdot X^{ks-(ns+1)}$ and sends $[P_A(\tau)]_1$ to the verfier.

Round 11 (Verifier). The verifier proceeds as follows:

(1) Checks that A encodes the correct values:

$$e([A(\tau)]_1, [T(\tau)]_2 + \delta[\tau]_2) = e([Q_A(\tau)]_1, [Z_{\mathbb{W}}(\tau)]_2) \cdot e([M(\tau)]_1 - \beta[A(\tau)]_1, [1]_2)$$

(2) **Degree Check:** If $n > k$, check

$$e([B_0(\tau)]_1, [\tau^{ns-ks-1}]_2) = e([P_B(\tau)]_1, [1]_2)$$

else if $k > n$, check

$$e([A_0(\tau)]_1, [\tau^{ks-ns-1}]_2) = e([P_A(\tau)]_1, [1]_2)$$

(3) Samples random $\gamma \in \mathbb{F}$ and sends them to the prover.

***In Rounds 12-15**, the prover gives a correctness proof of B at a random challenge. As we mentioned before, we combine this check with the correctness checks of eqs 3 and 4 to reduce the number of pairing checks and give the KZG-opening proofs corresponding to all the involved polynomials evaluated at the random challenge.*

Round 12 (Prover). The prover sends $b_{0,\gamma} = B_0(\gamma)$, $f_\gamma = F(\gamma)$, $\ell_\gamma = L(\gamma), a_0 = A(0)$, $\ell_{\gamma,\nu} = L(\nu\gamma)$, $q_{\gamma,L} = Q_L(\gamma)$, $d_\gamma = D(\gamma)$, $q_{\gamma,D} = Q_D(\gamma)$. to the verifier[16].

Round 13 (Verifier). Verifier samples a random $\eta \in \mathbb{F}$ and sends to the prover.

Round 14 (Prover). The prover computes the following: $P(X) = L(X\nu) + \eta L(X) + \eta^2 Q_L(X) + \eta^3 D(X) + \eta^4 Q_D(X) + \eta^5 B_0(X) + \eta^6 F(X) + \eta^7 Q_B(X)$ and

$$H_P(X) = \frac{P(X) - p_\gamma}{X - \gamma}$$

[15] Given $q_{i,1}, q_{i,2}$, and $[\psi_i^{\mathbb{W}}(\tau)]_1$, for each $i \in [ns]$, we can slightly modify the step 2 of lemma 4 to obtain $[Q_A(\tau)]_1$ and the commitment of the remainder polynomials: compute a linear combination of the quotients and remainders corresponding to the division of $A(X)T(X)$ and $A(X)X$ by $Z_{\mathbb{W}}(X)$.

[16] Given the sparse representation of A, the prover can compute $A(0)$ in time $O(ks)$, given the pre-computed Lagrange evaluations at 0.

where $p_\gamma = \ell_{\gamma,v}+\eta\ell_\gamma+\eta^2 q_{\gamma,L}+\eta^3 d_\gamma+\eta^4 q_{\gamma,D}+\eta^5 b_{0,\gamma}+\eta^6\cdot f_\gamma+\eta^7 q_{B,\gamma}$, for $b_\gamma = b_{0,\gamma}\cdot\gamma + B(0)$ and

$$q_{B,\gamma} = \frac{b_\gamma(f_\gamma+\beta+\delta\ell_\gamma)-1}{Z_{\mathbb{V}}(\gamma)}.$$

The prover then sends $[H_P(\tau)]_1$ and $[P(\tau)]_1$ to the verifier.

Round 15 (Verifier). The verifier proceeds as follows:

(1) Sets $b_0 = ns\cdot a_0/ks$.
(2) As part of checking the correctness of B, it computes $Z_{\mathbb{V}}(\gamma) = \gamma^{ks}-1$, $b_\gamma = b_{0,\gamma}\cdot\gamma + b_0$ and
$$q_{B,\gamma} = \frac{b_\gamma(f_\gamma+\beta+\delta\ell_\gamma)-1}{Z_{\mathbb{V}}(\gamma)}$$
(3) Computes $p_\gamma = \ell_{\gamma,v}+\eta\ell_\gamma+\eta^2 q_{\gamma,L}+\eta^3 d_\gamma+\eta^4 q_{\gamma,D}+\eta^5 b_{0,\gamma}+\eta^6\cdot\mathsf{com}+\eta^7 q_{B,\gamma}$.
(4) Checks the following:
$$e([H_p(\tau)]_1,[\tau]_2) = e([P(\tau)]_1 - p_\gamma + \gamma[H_p(\tau)]_1.[1]_2)$$
$$e([A(\tau)]_1-[a_0]_1,[1]_2) = e([A_0(\tau)]_1,[\tau]_2)$$
$$(\gamma^k-1)(\ell_{\gamma,v}-\omega\ell_\gamma) - Z_{\mathbb{V}}(\gamma)q_{\gamma,L} = 0$$
$$\ell_\gamma - d_\gamma - Z_{\mathbb{K}}(\gamma)q_{\gamma,D} = 0$$

B.3 Proof of Theorem 1

Proof. It is easy to check that our protocol is complete, assuming the completeness of the caulk sub-protocol.

Proof of Knowledge Soundness. Suppose $\mathcal{A}$ is an efficient algebraic adversary attacking the knowledge soundness of our protocol, as in Definition 2. Since $\mathcal{A}$ is algebraic, it will send a polynomial $F(X)\in\mathbb{F}_{<ks}[X]$ corresponding to com in Step 3) of the knowledge soundness game. Furthermore, corresponding to all the commitments in the protocol, $\mathcal{A}$ sends the corresponding polynomials of appropriate degree. Let Win denote the event that $\mathcal{A}$ wins the knowledge soundness game and Acc denote the event that the verifier accepts. Then, Win $\subset$ Acc and Acc implies that all the pairing checks in our protocol verify. By Lemma 1, this means that the corresponding ideal checks will also verify, except with a $\mathsf{negl}(\lambda)$-probability. Assuming this, we want to prove that for each $i\in[0,k-1]$, there exists $j\in[0,n-1]$ such that, for each $q\in[0,s-1]$, $F(\nu^{is+q}) = T(\omega^{js+q})$.

If all the ideal pairing checks verify, then the following holds.

- By the security of KZG commitments, the pairing checks in Round 15 implies that except with probability $ks/|\mathbb{F}|$ over $\gamma,\eta\in\mathbb{F}$, it holds that $\ell_{\gamma,v} = L(\gamma\nu)$, $\ell_\gamma = L(\gamma)$, $q_{\gamma,L} = Q_L(\gamma)$, $d_\gamma = D(\gamma)$, $q_{\gamma,D} = Q_D(\gamma)$, $b_{0,\gamma} = B_0(\gamma)$, $f_\gamma = F(\gamma)$ and $q_{B,\gamma} = Q_B(\gamma)$.
- This implies that the checks in Round 15, Step 4) at the random point γ implies that the following equations hold:
$$B(X)(\beta+F(X)+\delta L(X))-1 = Z_{\mathbb{V}}(X)Q_B(X)$$
$$(X^k-1)(L(X\nu)-\omega L(X)) = Z_{\mathbb{V}}(X)Q_L(X)$$
$$L(X)-D(X) = Z_{\mathbb{K}}(X)Q_D(X)$$
- By the knowledge soundness of the caulk sub-protocol (c.f. [62, Theorem 4]), Rounds 3-8 in our protocol guarantee that L is well-formed, i.e., for each $i\in[0,k-1]$, $D(\nu^{is})\in\{\mu^0,\cdots,\mu^{n-1}\}$, for $\mu=\omega^s$.
- Since the verifier obtains $[\tau^{ks-ns-1}]_2$, $[\tau^{ns-ks-1}]_2$ from the srs, the degree pairing checks in Round 11 imply the the following:
  - If $n>k$, then $deg(B_0)\le ks-2$, and if $k>n$, then $deg(A_0)\le ns-2$.
  - In either case above, the other polynomial also satisfies the degree check because the commitment of the corresponding power of τ will be the highest $\mathbb{G}_1$-power in srs. This is also why no degree check is needed if $k=n$. Furthermore, since $B(X) = B_0(X)X + b_0$ for $b_0 = nsa_0/ks$, $deg(B)<ks$.
- In Round 11, we check the following equation at τ:
$$A(X)(\beta+T(X)+\delta X)-M(X) = Z_{\mathbb{W}}(X)Q_A(X)$$
This implies that $A(\omega^i) = \frac{M(\omega^i)}{\beta+T(\omega^i)+\delta\omega^i}$ for each $i\in[ns]$. Furthermore, since B is well-formed, we know that $\sum_{i\in[ks]}B(\nu^i) = \sum_{i\in[ks]}\frac{1}{\beta+F(\nu^i)+\delta L(\nu^i)}$. Further, by Lemma 3, $\sum_{i\in[ks]}B(\nu^i) = ks\cdot b_0$ and $\sum_{i\in[ns]}A(\omega^i) = ns\cdot a_0$. Since, in the protocol b_0 is set such that $ks\cdot b_0 = ns\cdot a_0$, we have that $\sum_{i\in[ns]}\frac{M(\omega^i)}{\beta+T(\omega^i)+\delta\omega^i} = \sum_{i\in[ks]}\frac{1}{\beta+F(\nu^i)+\delta L(\nu^i)}$. This implies that except with probability $(ks\cdot ns)/|\mathbb{F}|$ over $\beta\in\mathbb{F}$, the following will hold:
$$\sum_{i\in[ns]}\frac{M(\omega^i)}{X+T(\omega^i)+\delta\omega^i} = \sum_{i\in[ks]}\frac{1}{X+F(\nu^i)+\delta L(\nu^i)} \tag{5}$$
This implies that the following two equation checks on τ in Rounds 2 and 15,
$$(X^n-1)(M(X)-M(X/\omega)) = Z_{\mathbb{W}}(X)Q_M(X)$$
$$A(X)-a_0 = A_0(X)\cdot X$$
will guarantee that these equations hold and that M is guaranteed to be well-formed[17]. Thus, by Lemma 6, eq. 5 implies that for each $i\in[ks]$, there exists some $j\in[ns]$ such that $F(\nu^i)+\delta L(\nu^i) = T(\omega^j)+\delta\omega^j$.
- Thus, by the definition of polynomial L, this implies that for each $i\in[0,k-1]$, there exists a $j\in[0,n-1]$ such that for each $q\in[0,s-1]$, $F(\nu^{is+q})+\delta\omega^{js+q} = T(\omega^{js+q})+\delta\omega^{js+q}$, i.e., $F(\nu^{is+q}) = T(\omega^{js+q})$.

Thus, the event Acc that the verifier accepts implies that all the ideal pairing checks will verify except with a $\mathsf{negl}(\lambda)$ probability. By the above implications, this in turn implies that for each $i\in[0,k-1]$, there exists a $j\in[0,n-1]$ such that for each $q\in[0,s-1]$, $F(\nu^{is+q}) = T(\omega^{js+q})$, except with a $\mathsf{negl}(\lambda)$ probability. This proves the knowledge soundness of our protocol.

Efficiency. The efficiency of gen follows from Lemma 4. The additional multiplicative log-factor is due to the use of FFTs to compute the quotient polynomials. By Lemma 4 it is guaranteed that the online prover cost of our protocol is dominated by the following two costs: first, the instantiation of Lemma 5 uses $O(ks\log n)$ $\mathbb{G}_1$- and $\mathbb{F}$-operations (details in Appendix D); second, the FFT computation for computing the quotient polynomials, on $\mathbb{V}$ requires $O(ks\log ks)$ $\mathbb{F}$-operations. It is easy to see that our protocol's proof size and verifier cost are $O(1)$.

□

[17] if M was not correctly formed, and the pairing check of round 2 was set to be true by $\mathcal{A}$, then the equality guaranteed by eq. 5 will not hold.

B.4 Adding Zero-Knowledge

Our segment-lookup protocol as described in the previous section, does not achieve *zero-knowledge* (i.e., it does not hide the segments encoded using polynomial $F(X)$). However, as is the case with most existing efficient SNARKs [17, 20, 25], our protocol can be easily modified to achieve zero-knowledge with the caveat that it *leaks the effective layer k of the input.* The main idea is to use randomized polynomial encodings (as opposed to unique polynomial encodings) when computing commitments to witness-dependent vectors. The reason why this simple modification suffices for ensuring zero-knowledge is because, throughout the protocol, the verfier only sees evaluations of these polynomial encodings at a few random points. When using higher-degree randomized polynomial encodings, these polynomial evaluations do not leak any information about the encoded vector.

In more detail, if for instance the verifier receives α evaluations of polynomial $F(X)$ in our segment-lookup protocol, then it suffices for the prover to commit to and work with a randomized polynomial $\hat{F}(X) = F(X) + R(X) \cdot Z_{\mathbb{V}}(X)$ instead of $F(X)$, where $R(X)$ is a random polynomial of degree $\alpha - 1$. Our segment-lookup protocol will achieve zero-knowledge if all witness-dependent polynomials in the above protocol are randomized in a similar manner. For simplicity of presentation, we chose to present the protocol in the previous section without the zero-knowledge property. Since the primary focus of our work was to reduce the prover run-time, this allowed us to highlight the main technical ideas without having to deal with unnecessarily complex notation.

C $\mathcal{S}\mathtt{ublon}\mathcal{K}$ PROTOCOL

C.1 Post-Processing F_Y Polynomials

In this section, we describe how the F_Y polynomials, can be *post-processed* to obtain a verifier pre-processing identical to that in $\mathcal{P}\mathtt{lon}\mathcal{K}$ for the activated sub-circuit $\widetilde{C}$. Let $\{\widetilde{\mathsf{q_M}}, \widetilde{\mathsf{q_L}}, \widetilde{\mathsf{q_R}}, \widetilde{\mathsf{q_O}}, \widetilde{\mathsf{q_C}}, \widetilde{\mathsf{S}_{\sigma 1}}, \widetilde{\mathsf{S}_{\sigma 2}}, \widetilde{\mathsf{S}_{\sigma 3}}\}$ denote the $\mathcal{P}\mathtt{lon}\mathcal{K}$ constraints for $\widetilde{C}$. We prove the following claims.

Selector Polynomials. For $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}\}$, we observe that the corresponding F_Y polynomials are exactly the selector polynomials $\{\widetilde{\mathsf{q_M}}, \widetilde{\mathsf{q_L}}, \widetilde{\mathsf{q_R}}, \widetilde{\mathsf{q_O}}, \widetilde{\mathsf{q_C}}\}$.

Claim 3. *For $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}\}$, $F_Y(X) = \widetilde{Y}(X)$.*

Proof Sketch. We sketch the proof for $\widetilde{\mathsf{q_M}}(X)$ and $F_{\mathsf{q_M}}(X)$, and the proof extends identically to the other polynomials in the claim. For any $i \in [0, k-1], j \in [s]$, it is the case that $\widetilde{q}_M(\nu^{is+j}) = q_M^{(\xi(i))}(\eta^j) = F_{\mathsf{q_M}}(\nu^{is+j})$, where the first equality follows from the fact that $\widetilde{\mathsf{C}} = (\mathsf{C}^{(\xi(i))})_{i \in [0,k-1]}$ and the definition of $q_M^{(\xi(i))}(X)$, and the second follows from the definition of $F_{\mathsf{q_M}}(X)$.

□

Permutation Polynomials. For $Y \in \{\mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$, we observe that the corresponding F_Y polynomials are a function of the selector polynomials $\{\widetilde{\mathsf{S}_{\sigma 1}}, \widetilde{\mathsf{S}_{\sigma 2}}, \widetilde{\mathsf{S}_{\sigma 3}}\}$. More formally,

Claim 4. $\forall a \in [3], \forall i \in [0, k-1], j \in [s], F_{\mathsf{S}_{\sigma a}}(\nu^{is+j}) \cdot \nu^{is} = \widetilde{\mathsf{S}_{\sigma a}}(\nu^{is+j})$

Proof Sketch. For any $i \in [0, k-1], j \in [s]$, consider $\widetilde{\mathsf{S}_{\sigma a}}(\nu^{is+j})$. As described, the permutation $\widetilde{\sigma}$ for C can be split into disjoint permutations $\{\widetilde{\sigma}^{(i)}\}_i$, where by the definition of $\mathcal{P}\mathtt{lon}\mathcal{K}$, $\widetilde{\sigma}^{(i)} : \mathcal{I}_i \to \mathcal{I}_i$ for $\mathcal{I}_i \coloneqq [is+1, (i+1)s] \cup [ks+is+1, ks+(i+1)s] \cup [2ks+is+1, 2ks+(i+1)s]$.

This in turn implies that $\widetilde{\mathsf{S}_{\sigma a}}(\nu^{is+j}) \in \{\nu^{is+\ell}, \nu^{is+\ell} \cdot k_1, \nu^{is+\ell} \cdot k_1\}$, where $\ell \in [s]$ and k_1 and k_2 are such that $k_1\mathbb{V}$ and $k_2\mathbb{V}$ are disjoint cosets of $\mathbb{V}$ (see $\mathcal{P}\mathtt{lon}\mathcal{K}$ [25] for details regarding the cosets). Further, ℓ is determined by $\sigma^{\xi(i)}(j)$, the permutation polynomial for $\mathsf{C}^{(\xi(i))}$. Therefore, we have for any $i \in [0, k-1], j \in [s]$,

$$\begin{aligned}\widetilde{\mathsf{S}_{\sigma a}}(\nu^{is+j}) &= \widetilde{\sigma}^{*(i)}(j) = \nu^{is}\sigma^{*(\xi(i))}(j) \\ &= \nu^{is}\mathsf{S}_{\sigma a}^{(\xi(i))}(\eta^j) = \nu^{is}F_{\mathsf{S}_{\sigma a}}(\nu^{is+j})\end{aligned}$$

□

Post-Processing the $\mathcal{P}\mathtt{lon}\mathcal{K}$ Permutation Polynomial. From our previous two claims, it is clear that while the (commitment of the) selector polynomials output from the segment-lookup protocol work as is, the same is *not* true of the permutation polynomial. But we will utilize Claim 4 to make the prover send the commitment to the correct permutation polynomial $\widetilde{\mathsf{S}_{\sigma a}}(X)$, and prove the relation to $F_{\mathsf{S}_{\sigma a}}(X)$. To this end, we define a new polynomial

$$U(X) \coloneqq \sum_{i=0}^{k-1} \nu^i \sum_{j=1}^{s} \psi_{is+j}^{\mathbb{V}}(X).$$

From Claim 4, we have that $i \in [0, k-1], j \in [s]$

$$F_{\mathsf{S}_{\sigma a}}(\nu^{is+j}) \cdot U(\nu^{is+j}) - \widetilde{\mathsf{S}_{\sigma a}}(\nu^{is+j}) = 0.$$

Since the above holds for all $\nu \in \mathbb{V}$, it can be represented by the following polynomial check,

$$F_{\mathsf{S}_{\sigma a}}(X) \cdot U(X) - \widetilde{\mathsf{S}_{\sigma a}}(X) = Z_{\mathbb{V}}(X) Q_{\mathsf{S}_{\sigma a}}(X)$$

Given commitments to the polynomials above, the verifier will check that the equation is satisfied at a random point, and then proceed to invoking $\mathcal{P}\mathtt{lon}\mathcal{K}$ using the commitment to $\widetilde{\mathsf{S}_{\sigma a}}(X)$. We note that the commitment to $U(X)$ will be produced as a part of the pre-processing since it is input independent.

C.2 $\mathcal{S}\mathtt{ublon}\mathcal{K}$ Protocol Description

We finally describe our full $\mathcal{S}\mathtt{ublon}\mathcal{K}$ protocol in Figure 4.

We state our theorem relative to the costs of the segment-lookup and $\mathcal{P}\mathtt{lon}\mathcal{K}$ protocols. Specifically, $\mathsf{Time}_{\mathsf{P}}^{\mathsf{segmentLookup}}(n, k, s)$ denotes the prover time for the segment-lookup protocol when parameterized by n, k and s. Similarly $\mathsf{Time}_{\mathsf{P}}^{\mathcal{P}\mathtt{lon}\mathcal{K}}(ks)$ denotes the prover time when $\mathcal{P}\mathtt{lon}\mathcal{K}$ is run on a circuit of size $O(ks)$. Verifier time and proof size are denoted in an analogous manner.

Theorem 3. *Given a $(s, \bar{k}, n, \{\mathsf{C}_i\}_{i=1}^n)$ layered branching circuit* C *such that can be partitioned into n sub-circuits of size s such that the execution path has length $O(ks)$, the above protocol is a pre-processing SNARK in the Algebraic Group Model for $\mathscr{C}$ induced by* C *such that the following properties hold:*

- **Prover Time:** $\mathsf{Time}_{\mathsf{P}}^{\mathsf{segmentLookup}}(n, k, s) + \mathsf{Time}_{\mathsf{P}}^{\mathcal{P}\mathtt{lon}\mathcal{K}}(ks) + O(ks \log(ks))$ $\mathbb{G}_1$ *and* $\mathbb{F}$ *operations*
- **Verifier Time:** $\mathsf{Time}_{\mathsf{V}}^{\mathsf{segmentLookup}}(n, k, s) + \mathsf{Time}_{\mathsf{V}}^{\mathcal{P}\mathtt{lon}\mathcal{K}}(ks) + 3$ *Pairings.*

$\mathsf{PreProcess}(\mathsf{C} = (k, \{\mathsf{C}^{(i)}\}_{i\in[n]}))$

(1) Define polynomials $(T_{\mathsf{q_M}}(X), T_{\mathsf{q_L}}(X), T_{\mathsf{q_R}}(X), T_{\mathsf{q_O}}(X), T_{\mathsf{q_C}}(X), T_{\mathsf{S}_{\sigma 1}}(X), T_{\mathsf{S}_{\sigma 2}}(X), T_{\mathsf{S}_{\sigma 3}}(X))$ as described in Section 4.1.
(2) Run pre-processing for the segment-lookup, i.e. for each $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}, \mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$,
$$\mathsf{pre}_Y = \mathsf{gen}(n, k, s, T_Y).$$
(3) Compute $U(X) \coloneqq \sum_{i=0}^{k-1} \nu^i \sum_{j=1}^{s} \psi^{\mathbb{V}}_{is+j}(X)$
(4) Output $\mathsf{pre} \coloneqq \{\mathsf{pre}_Y\}_Y, [U(\tau)]_2$.

$\mathcal{S}\mathfrak{ublon}\mathcal{K}(\mathsf{pre}, \phi, \mathsf{C}, \mathbb{V}, \mathbb{W})$

(1) P parses the execution path $\widetilde{\mathsf{C}} = \{\widetilde{\mathsf{C}}^{(i)}\}_{i\in[k]}$, and computes the function $\xi : \mathbb{F}^m \to [1, n]^k$ (on statement and witness pair $x = (\phi, w)$), that maps $\widetilde{\mathsf{C}}^{(i)}$ to $\mathsf{C}_{\xi_x(i)}$.
(2) P uses ξ to compute $F_Y(X)$ for all $Y \in \{\mathsf{q_M}, \mathsf{q_L}, \mathsf{q_R}, \mathsf{q_O}, \mathsf{q_C}, \mathsf{S}_{\sigma 1}, \mathsf{S}_{\sigma 2}, \mathsf{S}_{\sigma 3}\}$.
(3) P and V run $\mathsf{segmentLookup}^{\otimes \ell}(T_{\mathsf{q_M}}, T_{\mathsf{q_L}}, T_{\mathsf{q_R}}, T_{\mathsf{q_O}}, T_{\mathsf{q_C}}, T_{\mathsf{S}_{\sigma 1}}, T_{\mathsf{S}_{\sigma 2}}, T_{\mathsf{S}_{\sigma 3}})$ with verifier output $([F_{\mathsf{q_M}}(\tau)]_1, [F_{\mathsf{q_L}}(\tau)]_1, [F_{\mathsf{q_R}}(\tau)]_1, [F_{\mathsf{q_O}}(\tau)]_1, [F_{\mathsf{q_C}}(\tau)]_1, [F_{\mathsf{S}_{\sigma 1}}(\tau)]_1, [F_{\mathsf{S}_{\sigma 2}}(\tau)]_1, [F_{\mathsf{S}_{\sigma 3}}(\tau)]_1)$.
(4) P sends $[\widetilde{\mathsf{S}_{\sigma 1}}(\tau)]_1, [\widetilde{\mathsf{S}_{\sigma 2}}(\tau)]_1, [\widetilde{\mathsf{S}_{\sigma 3}}(\tau)]_1$ to V.
(5) V sends a random $\gamma \in \mathbb{F}$ to P.
(6) P computes for each $a \in [3]$, $[Q_{\sigma_a}(\tau)]_1$ such that
$$U(X) \cdot F_{\mathsf{S}_{\sigma a}}(X) - \widetilde{\mathsf{S}_{\sigma a}}(X) = \mathbb{Z}_{\mathbb{V}}(X) \cdot Q_{\sigma_a}(X)$$
(7) V checks if the following holds for each $a \in [3]$,
$$e([F_{\mathsf{S}_{\sigma a}}(\tau)]_1, [U(\tau)]_2) \cdot e([\widetilde{\mathsf{S}_{\sigma a}}(\tau)]_1, [1]_2)^{-1} = e([Q_{\sigma_a}(\tau)]_1, [\mathbb{Z}_{\mathbb{V}}(\tau)]_2)$$
(8) P and V run $\mathcal{P}\mathfrak{lon}\mathcal{K}$ $(\mathsf{P}_{\mathcal{P}\mathfrak{lon}\mathcal{K}}, \mathsf{V}_{\mathcal{P}\mathfrak{lon}\mathcal{K}})$ with pre-processed inputs $([F_{\mathsf{q_M}}(\tau)]_1, [F_{\mathsf{q_L}}(\tau)]_1, [F_{\mathsf{q_R}}(\tau)]_1, [F_{\mathsf{q_O}}(\tau)]_1, [F_{\mathsf{q_C}}(\tau)]_1, [\widetilde{\mathsf{S}_{\sigma 1}}(\tau)]_1, [\widetilde{\mathsf{S}_{\sigma 2}}(\tau)]_1, [\widetilde{\mathsf{S}_{\sigma 3}}(\tau)]_1)$
(9) V accepts if all checks pass.

Figure 4: The $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ Protocol

Proof Size: $\mathsf{Size}_{\mathsf{V}}^{\mathsf{segmentLookup}}(n, k, s) + \mathsf{Size}_{\mathsf{V}}^{\mathcal{P}\mathfrak{lon}\mathcal{K}}(ks) + 13\,\mathbb{G}_1$ *elements.*

Except with negligible probability (over the choice of γ), given that the verifier check in Step 7 succeeds, the commitments sent by the prover in Step 4 must be the correct polynomial. Thus the security of the scheme follows directly from the security of the underlying schemes.

Plugging in the asymptotic costs of the underlying protocol, we have the following corollary for Thm 3, which immediately implies Thm 2.

Corollary 1. *The asymptotic costs for the protocol in Theorem 3 is $O(ks \cdot (\log(ks) + \log(n)))$ $\mathbb{G}_1$- and $\mathbb{F}$-operations for prover cost, and $O(1)$ for verifier cost and proof size.*

Efficiency of $\mathcal{S}\mathfrak{ublon}\mathcal{K}$. The $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ prover invokes the segment-lookup protocol in parallel on each of the 8 types of polynomials needed for running the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol. This contributes to the first component, $\mathsf{Time}_{\mathsf{P}}^{\mathsf{segmentLookup}}(n, k, s)$, of the prover cost. Secondly, to compute quotient polynomials and their commitments in Step 6 of Figure 4, the prover uses FFT, which contributes to the third component, $O(ks \log(ks))$, of the prover cost. Finally, the prover runs the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ prover in Step 8, which contributes to the second component, $\mathsf{Time}_{\mathsf{P}}^{\mathcal{P}\mathfrak{lon}\mathcal{K}}(ks)$, of the prover cost. By Thm 1, we know that $\mathsf{Time}_{\mathsf{P}}^{\mathsf{segmentLookup}}(n, k, s) = O(ks \cdot (\log(ks) + \log(n)))$ $\mathbb{G}_1$- and $\mathbb{F}$-operations, which dominates the cost of the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ prover. Thus, our $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ prover cost is $O(ks \cdot (\log(ks) + \log(n)))$ $\mathbb{G}_1$- and $\mathbb{F}$-operations.

Allowing Arbitrary Number of Effective Layers. So far in this section, we assume that the effective number of layers k is fixed in advance. However, as discussed in Section 2.1, k depends on the input to the circuit C. Our protocol can be easily generalized to handle this case. In particular, let $\bar{k}$ be the maximum number of layers in C. We can easily modify the pre-processing algorithm in $\mathcal{S}\mathfrak{ublon}\mathcal{K}$ to take as input $\bar{k}$, instead of k as follows: The modified pre-processing algorithm does the computation dependent on n, as described in Figure 4. For the remaining computation in the pre-processing phase that depends on k, we compute this for every $k' < \bar{k}$. In the online phase, once the prover learns the value of k upon evaluating the circuit on a given statement and witness, he can communicate this k to the verifier. For the rest of the protocol, both the prover and verifier work with the pre-processing corresponding to $k' = k$ and ignore the rest.

D CAULK SUB-PROTOCOL: MULTI-UNITY PROOF

For the sake of completeness, we give the full protocol description of [63, Figure 5] whose informal description was given in Section B.1. We require the following variant of bivariate KZG commitment from [63] for the sub-protocol.

KZG for Bivariate Polynomials. For a bivariate polynomial $P(X, Y)$ with degree up to $d_1 - 1$ in X and $d_2 - 1$ in Y, this protocol requires a universal setup with $d_1 d_2$ powers. The setup corresponding to the univariate KZG itself can be used for this. To commit to $P(X, Y)$, using the commit algorithm of univariate KZG, one can commit to $[P(\tau^{d_2}, \tau)]_1$. Opening this commitment requires two steps: first, partially open $P(X, Y)$ at some $X = \alpha$ to a commitment $[P(\alpha, \tau)]_1$. The partial proof is given by $[H_\alpha(\tau^{d_2}, \tau)]_1$, where $H_\alpha(X, Y) = \frac{P(X,Y) - P(\alpha,Y)}{X - \alpha}$; second, fully evaluate $P(\alpha, Y)$ at $Y = \beta$ via standard univariate KZG proof with a degree bound of $d_2 - 1$ on $[P(\alpha, \tau)]_1$.

The main protocol takes as input the commitment $[u_0]_1$ and polynomial $U_0(X)$, and proves that the commitment indeed corresponds to $U_0(X)$ that encodes $(D(1), D(\nu^s), \ldots, D(\nu^{(k-1)s}))$, and checks that for each $i \in [0, k-1]$, $D(\nu^{is}) \in \{\mu^0, \cdots, \mu^{n-1}\}$, for $\mu = \omega^s$. As explained in the informal description, this is equivalent to proving that the following aggregated equation holds:
$$(U_0^2(X)\Delta_1(Y) + \sum_{j=2}^{\log n} U_{j-1}^2(X)\Delta_j(Y)) - (\sum_{j=1}^{\log n - 1} U_j(X) \cdot \Delta_j(Y) + \mathsf{id}(X)\Delta_{\log n}(Y)) = Z_{\mathbb{V}}(X)Q_2(X, Y)$$

for some polynomial $Q_2(X, Y)$. We describe the protocol details below, which helps check the above equation at a random point (α, β). On a high level, each round of the protocol does the following:

- Checking the above equation for some polynomial $Q_2(X, Y)$ at (α, β) is equivalent to showing that the following polynomial evaluates to 0 at $Y = \beta$:

$$P(Y) = (U_0^2(\alpha)\Delta_1(\beta) + \sum_{\ell=2}^{\log n} U_{\ell-1}^2(\alpha)\Delta_\ell(\beta) + Z_{\mathbb{S}}(\beta)(-Q_1(\beta) + Q_1(Y))) - (\sum_{\ell=1}^{\log n-1} U_\ell(\alpha)\Delta_\ell(\beta) + \mathsf{id}(\alpha)\Delta_{\log n}(\beta)) - Z_{\mathbb{V}}(\alpha)Q_2(\alpha, Y)$$

  for some polynomial $Q_1(Y)$. For this, the prover sends commitments and values needed to reconstruct $[P(\tau)]_1$ and provides a proof for opening P at 0 and β.
- The other observation used is that since Δ_ℓ's take 0/1 values, for each $Y \in \mathbb{S}$, it holds that:

$$(U_0^2(X)\Delta_1(Y) + \sum_{\ell=2}^{\log n} U_{\ell-1}^2(X)\Delta_\ell(Y)) = (U_0(X)\Delta_1(Y) + \sum_{\ell=2}^{\log n} U_{\ell-1}(X)\Delta_\ell(Y))^2$$

  In the protocol below, we denote $\bar{U}(X, Y) = \sum_{\ell=2}^{\log n} U_{\ell-1}(X)\Delta_\ell(Y)$ and $U(X, Y) = \bar{U}(X, Y) + U_0(X)\Delta_1(Y)$.
- Now, the prover takes the univariate polynomials corresponding to $\bar{U}$ and Q_2 and sends the commitments $[\bar{U}(\tau^{\log n}, \tau)]_1, [Q_2(\tau^{\log n}, \tau)]_1$ to the verifier. This is intended to be a bivariate commitment, but we want to reuse the srs containing only powers of τ.
- For the remaining rounds, the prover essentially sends the necessary commitments and opening proofs for the remaining polynomials corresponding to the equation above. Additional to the equation check, one additional check is done for the correctness of $\bar{U}$ and U (check $\bar{U}(X, 1) = 0$, and then enforce the degree check).

The protocol uses the srs from our protocol of section B.2. The common inputs for the prover and verifier is $[U_0(\tau)]_1$.

Round 1 (Prover → Verifier). The prover takes the input srs and $U_0(X)$ and samples $t_1, \cdots, t_{\log n} \leftarrow \mathbb{F}$ to compute:

(1) For $\ell = 1, \cdots, \log n$, $U_\ell(X) = \sum_{j=1}^{ks} (\mu^{i_j})^{2^\ell} \Delta_j(X) + t_\ell Z_{\mathbb{V}}(X)$.
(2) $U(X, Y) = \sum_{\ell=1}^{\log n} U_{\ell-1}(X)\Delta_\ell(Y)$.
(3) $\bar{U}(X, Y) = U(X, Y) - U_0(X)\Delta_1(Y)$.
(4) $Q_2(X, Y) = \sum_{\ell=1}^{\log n} \Delta_\ell(Y)Q_{2,\ell}(X)$, for $Q_{2,\ell}(X) = (U_{\ell-1}^2(X) - U_\ell(X))/Z_{\mathbb{V}}(X)$.

The prover sends $[\bar{U}(\tau^{\log n}, \tau)]_1, [Q_2(\tau^{\log n}, \tau)]_1$ to the verifier.

Round 2 (Verifier). The verifier sends a challenge $\alpha \in \mathbb{F}$.

Round 3 (Prover → Verifier). The prover computes $Q_1(Y) = (U^2(\alpha, Y) - \sum_{\ell=1}^{\log n} U_{\ell-1}^2(\alpha)\Delta_\ell(Y))/Z_{\mathbb{S}}(Y)$ and sends $[Q_1(\tau)]_1$ to the verifier.

Round 4 (Verifier). Sends challenge $\beta \in \mathbb{F}$.

Round 5 (Prover → Verifier). The prover does the following:

(1) Computes $P(Y) = (U^2(\alpha, \beta) - \bar{U}(\alpha, \beta\phi) + \mathsf{id}(\alpha)\Delta_{\log n}(\beta)) - Z_{\mathbb{V}}(\alpha)Q_2(\alpha, Y)$.
(2) Computes and sends $U_0(\alpha), [\bar{U}(\alpha, \tau)]_1, [Q_2(\alpha, \tau)]_1$, and their KZG opening proofs at α: π_1, π_2, π_3, respectively.
(3) Computes and sends $\bar{U}(\alpha, 1) = 0, \bar{U}(\alpha, \beta), \bar{U}(\alpha, \beta\phi)$ along with their KZG opening proof at $(1, \beta, \beta\phi)$: π_4.
(4) Computes and sends $P(\beta) = 0$, along with its opening proof at β: π_5.

Round 6 (Verifier). The verifier computes $[P(\tau)]_1 = (U_0(\alpha)\Delta_1(\beta) + \bar{U}(\alpha, \beta))^2 - [Q_1(\tau)]_1 Z_{\mathbb{S}}(\beta) - (\bar{U}(\alpha, \beta\phi) + \mathsf{id}(\alpha)\Delta_{\log n}(\beta)) - Z_{\mathbb{V}}(\alpha)[Q_2(\alpha, \tau)]_1$, and accepts if and only if all the KZG opening proofs $\pi_1, \cdots, \pi_5$ verify.

Overview of the Prover Cost Analysis. From the description above, we can see that the prover defines and commits to $\log n$ polynomials, each of degree ks, where the coefficients of these polynomials are a result of repeatedly squaring the coefficients of the original polynomial. Committing to these $\log n$ polynomials, each of degree ks requires the prover to perform $O(ks \log n)$ $\mathbb{G}_1$- and $\mathbb{F}$-operations. This becomes the dominant cost of this protocol (and hence our segment-lookup protocol).