Recursion over Public-Coin Interactive Proof Systems; Faster Hash Verification

Alexandre Belling, Azam Soleimanian, and Olivier Begassat

Consensys R&D, Consensys Inc.
{firstname.lastname}@consensys.net

Abstract. SNARK is a well-known family of cryptographic tools that is increasingly used in the field of computation integrity at scale. In this area, multiple works have introduced SNARK friendly cryptographic primitives - hashing, but also encryption and signature verification - which are used in many applications. Despite all the efforts to create cryptographic primitives that can be proved faster, it remains a major performance hole in practice. In this paper, we present a recursive technique that can improve the efficiency of the prover by an order of magnitude compared to proving MiMC hashes (a snark-friendly hash function, Albrecht et al. 2016) with a Groth16 (Eurocrypt 2016) proof. We use GKR (a well-known public-coin argument system by Goldwasser et al., STOC 2008) to prove the integrity of hash computations and embed the GKR verifier inside a SNARK circuit. The challenge comes from the fact that GKR is a public-coin interactive protocol, and applying Fiat-Shamir naively may result in worse performances than applying existing techniques directly. This is because Fiat-Shamir itself is involved with hash computation over a large string. We take advantage of a property that SNARK schemes commonly have to build a protocol in which for the Fiat-Shamir hashes have very short inputs. The technique we present is generic and can be applied to over most known SNARK scheme and any public-coin argument system in place of GKR. It outputs an augmented proof system combining the performances of the two.

Keywords: SNARK · Hash Verification · Proof Recursion · Proof Composition · GKR · Public-Coin · Fiat Shamir · So-Far Digest Model

Table of Contents

1 Introduction

Succinct Non-Interactive Argument of Knowledge (zk-)SNARKs are powerful cryptographic tools that allow a prover to convince a verifier that a relation (usually drawn from a large family) is satisfied with respect to public inputs [1], such that the verifier needs less time to verify the proof rather than redoing all the computation. In recent years, an ever-growing number of SNARK constructions have emerged, including [23], [2], [8], [14], [30], [36], [11], [32] with various security assumptions and performance trade-offs. SNARKs are also widely adopted in the blockchain world for their applications for privacy (zk-SNARKs) [24] and scalable computational integrity [10]. Several important applications of SNARKs involve proving the computation of numerous hashes: signature and Merkle proof verification, which usually becomes the main bottleneck in the runtime of the prover.

Hashing inside a SNARK a SNARK scheme typically works over arithmetic circuits and a prespecified finite field. On the other hand, common hash functions such as SHA256, Blake2 or Keccak typically work with unsigned integers and bitwise operations, since they are faster on CPUs. Subsequently, even though they can be embedded within an arithmetic circuit, it is inefficient for the runtime of the prover to do so. Due to this fact, numerous works - among which MiMC [1], Poseidon [22] - have proposed SNARK-friendly hash functions: functions that are more efficient to embed in an arithmetic circuit by 2 orders of magnitude. Our contribution essentially focuses on applying new techniques to speed up the verification of MiMC hash function, resulting in a speed-up of x35 (more details in Fig. 19) compared to directly verifying MiMC with Groth16 [23]. However, the techniques we present can also be applied to the Poseidon hash function [22]. We describe this in Supplement C.

The GKR protocol [20] produces linear time verifiable proofs for multiple parallel execution of a layered arithmetic circuit C. The works [33], [34], [37] extend the GKR [20] protocol and improve its performances. In particular, [35] describes a generalization of the GKR protocol for arbitrary arithmetic circuits with a directed acyclic graph structure, which - in practice - also has a faster prover than the original version of GKR. Additionally, it does not require any particular cryptographic operations (apart from the Fiat-Shamir hashes). From their construction, we distill in Section 3.1, a variant of GKR tailored for the MiMC keyed-permutations [1]. The construction of Section 3.1 can be seen as analog to the custom gates in Ultra-PLONK [18].

Composing GKR with other argument systems Hyrax [32] proposes to compile GKR using discrete logarithm (DLog) assumptions to obtain a zk-SNARK. Its purpose is to establish an argument system without a trusted setup. LegoSnark, [11] presents a generic framework which enables linking a statement proven using GKR (or more precisely Hyrax [32]) to other ones using possibly

[1] This can be done without revealing additional information, if the SNARK has additionally the zero-knowledge property

different argument systems. Our work differs from theirs by embedding the GKR verifier within another SNARK. Our contribution is a construction that allows us to combine the GKR protocol with a SNARK.

Proof recursion is a technique that consists in verifying a publicly verifiable non-interactive proof inside another argument system. This technique can be used for building incrementally verifiable computation (IVC), proof carrying data (PCD) and proof aggregation. [5] specifies how to instantiate proof-carrying data through recursion using a pairing-friendly cycle of elliptic curves. [13] proposes a post-quantum argument system that is compatible with recursion. [8] proposes a technique for proof-carrying data using a (possibly non-pairing-friendly) cycle of elliptic curves. [31] describes an argument system optimized for recursion. Our work extends this model by working on *public-coin* non-interactive argument systems.

Technical Aspects and Motivation

The GKR protocol is originally a multi-round public-coin interactive protocol that is transformed to its non-interactive version by applying the Fiat-Shamir (FS) transform [12, 4], being sound in the random oracle model.[2] At first glance, it may seem strange to use GKR for hash verification (proof of correct hash computation). In particular, since the Fiat-Shamir transform of GKR, itself requires hash computations to generate challenges. We highlight some technical details which may convince the reader of the relevance of using GKR for hash verification.

- Parallelization: GKR can be parallelized for the hash verification of many claims.
- Recursion: We recurse GKR inside a SNARK. Namely, a SNARK is applied over the verifier of GKR such that the GKR verifier checks the correct computation of all hashes (leveraging GKR for data-parallel computation) and SNARK proves the correct execution of the GKR verifier.
- Practical verification-time: Another key idea that makes GKR very interesting for recursion is that — although its verifier runs asymptotically in linear time – its practical runtime is a fraction of the runtime of the alleged computation itself. Regarding efficiency, not all circuits are equals for application of GKR. Generally speaking, as a gross estimation, layered circuits with a smaller width, a larger depth and lower-degree (at each layer) are more interesting. This makes hash functions based on the S-Box $x \rightarrow x^{\alpha}$ where α is small, such as Poseidon [22] and MiMC [1] excellent candidates for GKR. [3]

[2] Indeed, in [12] the authors proved that GKR is round-by-round sound which implies security against state restoration attack [12]. In [4] the authors proved that if a multi-round protocol is secure against such attack, then it is sound in the random oracle model. Finally, [15] directly discusses the soundness of sumcheck protocol in the random oracle model.

[3] It is also possible to use GKR for Boolean circuits, [19] gives an example on how to do so.

- Compressing the input for Fiat-Shamir: Instead of including the input-output of GKR in the Fiat-Shamir hashes (as it would be *normally* required), we pass a prover-generated commitment which is much shorter. We further can adjust Fiat-Shamir to be more efficient inside the circuit.
- Native arithmetic: The challenge here is to force the prover, possibly malicious, to correctly compute the aforementioned commitment. We discuss how to circumvent this challenge in a non-trivial and efficient way. Note that a more trivial solution is to embed the commitment inside the circuit of SNARK, which is far less efficient due to the need for non-native arithmetic (since the commitment is a group element). Our technique provides a separate constant-size proof, for the correct computation of the commitment (outside the circuit). This guarantees the soundness of the system as a whole.

Overview of our technique

We present a technique to recurse a public-coin single-round interactive protocol inside another SNARK. In public-coin protocols, the verifier sends random challenges to the prover, which we call "randomnesses". GKR is an instance of public-coin protocols.

Our methodology allows the prover to handle the randomnesses efficiently when compiling to a non-interactive protocol and recursing inside a SNARK. The naive way would simply be to apply the Fiat-Shamir transform over GKR, but this would be inefficient. Indeed, when using the Fiat-Shamir transform, the verifier, and the prover are required to hash the past transcript including the inputs of the verifier. If these inputs are very long, this leads to long hashes to be performed inside the SNARK circuit, and these are expensive. Let assume n alleged evaluation of the MiMC keyed-permutation $H_{mimc}(x, k) = y$, applying the Fiat-Shamir transform directly needs hashing all the input-output x, y, k and would result in a protocol where we hash at least $3n$ field elements to obtain the initial challenge : at least 3 times worse than directly verifying hashes inside the circuit. Our protocol instead applies the Fiat-Shamir transform over a short input provided by the prover and externalizes the relevant computations on how this short input was obtained, outside the circuit. Slightly more in detail, the information that we use to generate the randomness is a piece of computation (let us call it γ_v) already required in the SNARK verification. As an example, consider the Groth16 scheme, where the verifier must first compute γ, a multi-scalar multiplication (MSM) of the verification key and the public input (as $\gamma = \prod_i \mathsf{vk}_i^{x_i}$). Second, it uses γ alongside the proof given by the prover inside a pairing check. When we recurse GKR inside Groth16, we do so in such a way that all the inputs of the GKR protocol are included in the public inputs. We call γ_v, the "part" of γ associated with the GKR inputs. In our scheme, the prover computes and sends γ_v to the verifier and provides an argument of knowledge that it knows a witness for γ_v regarding the appropriate verification key. Note that the prover still has to prove the correct computation of γ_v, but the advantage of using γ_v for generating the randomness is that,

- the verifier does not have to compute γ_v.
- It is hard for the prover to change γ_v, since it would be used in the verification of SNARKs.

In particular, by the second property, we can give an argument system with constant-size proofs to argue the correct computation of randomness (which is indeed $H(\gamma_v, \ldots)$, computed outside the circuit). Since γ_v depends on the outer-layer SNARK scheme, we present two argument systems for the integrity of γ_v, separately for Groth16 [23] and PLONK [2] as the outer-SNARK.

Our contribution

Our contribution can be divided into two main parts; theory and implementation.

Theoretical aspects: Although the idea of the paper was initially motivated by hash verification through recursion over GKR, our theoretical results are general and can be used for recursion over any public-coin interactive argument system. More precisely, we present a compiler that receives a single-round public-coin argument system and a SNARK (where it internally generates the required randomness based on γ_v) and outputs an efficient recursion system. We build our compiler step-by-step, and we analyze the security of each step separately. We prove that if the inputs to the compiler, and the argument of knowledge (AOK) for the correct computation of randomness satisfy the common security notion (knowledge-soundness), then the output of the compiler is also secure.

Implementation aspects: In our implementation, we use Groth16 as the outer-layer SNARK and recurse it over GKR with MiMC as the hash function. To improve efficiency, we use the custom gates specified in Fig. 1 and include all optimizations of [35] and [34] on the sumcheck protocol and GKR. Our implementation is in Golang and is optimized for massive parallelism (benchmarked on 96 physical cores). We expand further on that matter in Fig. 19.

Moreover, we use a realization of Fiat-Shamir that is more convenient to work with for our construction. We call such realization as Fiat-Shamir in the *so-far digest model*, the counterpart of the common so-far transcript model (see Supplement J), where instead of hashing the transcript, we hash the randomness of the previous round and the last message of the prover. This improves the efficiency overall since the Fiat-Shamir hash computation is done in the circuit (particularly, this realization avoids the hashing of public parameters inside the circuit). Working in this model can still be of independent interest for protocols that have some limitations in the choice of hash functions to instantiate the random oracle. Moreover, in Supplement J, we demonstrate that applying the Fiat-Shamir transform in the *so-far digest* model is sound if it is sound to do it in the *so-far transcript* model.

2 Background

2.1 Notations

We say $f(x) = \omega(g(x))$, if and only if $\lim_{x\to\infty} f(x)/g(x) = \infty$. Let λ denote the security parameter. We write $f(\lambda) \approx h(\lambda)$ when $|f(\lambda) - h(\lambda)| = \lambda^{-\omega(\lambda)}$ for two functions $f, h : N \to [0, 1]$. Then if $f(\lambda) \approx 0$ we say that f is negligible, and if $f(\lambda) \approx 1$ we say that f is overwhelming. We write $y \leftarrow A(x)$ to show that the algorithm A outputs y on input x. Through the paper, we assume that all the algorithms are probabilistic polynomial time (p.p.t.). By $x \leftarrow\!\$\, X$, we mean that the element x is chosen uniformly at random from the set X. The notation $[n]$ stands for the set $\{1, \ldots, n\}$.

To define a security notion, we may define a counterpart game $\mathbf{G}_{\mathcal{A}}$ as

$$\mathbf{G}_{\mathcal{A}} = (\text{wining condition, game interactions})$$

We say that the adversary $\mathcal{A}$ fails (or its advantage in $\mathbf{G}_{\mathcal{A}}$ is negligible) if,

$$\Pr[\text{ Wining condition : Game interactions for } \mathcal{A}] \approx 0$$

Groups. $\mathbb{G}$ denotes a cyclic group of prime order. If $\mathbb{G}$ is of order p, $g \in \mathbb{G}$, and $x \in \mathbb{Z}_p$, then g^x denotes the scalar multiplication. For a list of n scalars of $\mathbb{Z}_p$ and group elements of $\mathbb{G}$, the multi-scalar multiplication (MSM) is $\prod_{i\in[n]} {g_i}^{x_i}$. When it is clear that g is a generator of $\mathbb{G}$, we may use the notation $[x]$ for g^x.

Definition 1 (Bilinear Groups). *A bilinear group is a tuple $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ such that: $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ all cyclic groups, have prime order p, $e(\mathbb{G}_1, \mathbb{G}_2) \to \mathbb{G}_T$ is a bilinear, non-degenerate map that is efficiently computable. Also, throughout this work, g_1, g_2, g_T implicitly denotes generators of, respectively, $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ such that $e(g_1, g_2) = g_T$.*

The argument systems Groth16 [23] and PLONK [2] are proved to be secure under the q-DLog assumption (Supplement A, Definition 15) and in the Algebraic Group Model (AGM), formally defined as follows.

Definition 2 (Algebraic adversary in an SRS-based[4] protocol [7]). *An algebraic adversary $\mathcal{A}$ is a p.p.t algorithm which such that, whenever $\mathcal{A}$ outputs an element $A = [a]_i \in \mathbb{G}_i$, cA also outputs its linear combination based on srs_i, namely, a vector v of scalars such that $A = \langle v\ ,\ \mathsf{srs}_i\rangle = \sum_j v_j \cdot \mathsf{srs}_{ij}$, where srs_i is part of srs belonging to the group $\mathbb{G}_i$. We call such a representation as the linear-combination representation (LC-representation) of A.*

[4] The structured reference string (SRS) is the set of public parameters generated by the trusted setup with a special structure

2.2 Argument of Knowledge

We define $\mathcal{R}_\lambda$ to be a relation generator (i.e. $(\mathcal{R}, z) \leftarrow \mathcal{R}_\lambda$) such that $\mathcal{R}$ is a polynomial time decidable binary relation. For $\mathcal{R}(x; w)$, we call x as the statement and w as the witness. The relation generator may also output some side information z which will be given to the adversary. When we have several families of relations, we may show their relation generator as $\mathcal{R}_{F_i}$ (rather than $\mathcal{R}_\lambda$). We show the set of true statements by $\mathcal{L}_\mathcal{R} = \{x \ : \exists \ w \ \ \mathcal{R}(x; w) = 1\}$. The definitions is this section are mainly borrowed from [23].

Definition 3 (*non-interactive* Argument for $\mathcal{R}_\lambda$). *A Non-Interactive Argument for $\mathcal{R}_\lambda$ is a tuple of three p.p.t. algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *defined as follows,*

- $\sigma \leftarrow \mathsf{Setup}(\mathcal{R})$*: on input $\mathcal{R} \leftarrow \mathcal{R}_\lambda$, it generates a reference string σ. All the other algorithms implicitly receive the relation $\mathcal{R}$.*
- $\pi \leftarrow \mathsf{Prove}(\sigma, x, w)$*: it receives the reference string σ and for $\mathcal{R}(x; w) = 1$ it outputs a proof π.*
- $1/0 \leftarrow \mathsf{Verify}(\sigma, x, \pi)$*: it receives the reference string σ, the statement x and the proof π and returns 0 (reject) or 1 (accept).*

For the above argument system, we define the following security requirements.

Definition 4 (Completeness). *It says that given a true statement $x \in \mathcal{L}_\mathcal{R}$, the prover can convince the honest verifier; for all $\lambda \in \mathbb{N}$, $\mathcal{R} \in \mathcal{R}_\lambda, x \in \mathcal{L}_\mathcal{R}$:*

$$\Pr\left[1 = \mathsf{Verify}(\sigma, x, \pi) \ : \ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}), \ \pi \leftarrow \mathsf{Prove}(\sigma, x, w)\right] = 1$$

Definition 5 (Soundness). *It says that it is not possible to convince the verifier for the wrong statements. For any non-uniform p.p.t. adversary $\mathcal{A}$ we have,*

$$\Pr[1 = \mathsf{Verify}(\sigma, x, \pi) \wedge x \notin \mathcal{L}_\mathcal{R} \ : \ (\mathcal{R}, z) \leftarrow \mathcal{R}_\lambda, \ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}), \ (x, \pi) \leftarrow \mathcal{A}(\sigma, z)] \approx 0$$

Definition 6 (*non-interactive* Knowledge-Soundness). *It strengthens the notion of soundness by adding an extractor that can compute a witness from a given valid proof. The extractor gets full access to the adversary's state, including any random coins. Formally, for any non-uniform p.p.t adversary $\mathcal{A}$ there exists a non-uniform (expected polynomial time) extractor $\mathcal{X}_\mathcal{A}$ such that:*

$$\Pr\left[1 = \mathsf{Verify}(\sigma, x, \pi) \ \wedge \mathcal{R}(x; w) = 0 : \begin{array}{c}(\mathcal{R}, z) \leftarrow \mathcal{R}_\lambda, \ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}), \\ ((x, \pi), w) \leftarrow (\mathcal{A} \parallel \mathcal{X}_\mathcal{A})(\sigma, z)\end{array}\right] \approx 0$$

The advantage of adversary in the knowledge-soundness game (the probability on the left side) is called as knowledge-error.[5]

[5] Although, we only use the notion of knowledge-soundness throughout this work. The reader should be aware, a more general notion exists: witness-extended emulation.[6] Fortunately, in [26] Lindell shows that knowledge-soundness implies witness-extended emulation. Thus, for simplicity, we restrict ourselves to study knowledge-soundness of the protocols we describe.

An analog notion of knowledge-soundness can be defined for an interactive protocol. In this context, the definition is identical except that the extractor is only given black-box access to the transcript but is nonetheless allowed to rewind it up to any point in the interaction and to send the arbitrary messages.

Definition 7 (*interactive* Knowledge-Soundness). *An interactive argument system $(\mathcal{P}, \mathcal{V})$ has knowledge-soundness if for all (p.p.t. non-uniform) prover adversaries $\mathcal{A}$ there exists an (expected polynomial time) extractor $\mathcal{X}_{\mathcal{A}}$ with oracle access to $\mathcal{A}$, allowed to rewind $\mathcal{A}$ to any point in the interaction and to send it arbitrary messages such that the knowledge error in the following is negligible.*

$$\Pr\left[\begin{array}{l} 1 = \mathsf{Verify}(\sigma, x, \pi) \ \wedge \mathcal{R}(x; w) = 0 : \\ \end{array} \begin{array}{c} (\mathcal{R}, z) \leftarrow \mathcal{R}_\lambda, \ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}), \\ x \leftarrow \mathcal{A}(\sigma, z) \\ \pi \leftarrow \textit{Transcript} \ , \ w \leftarrow \mathcal{X}_{\mathcal{A}}^{\mathcal{O}}(\sigma, z, x) \end{array}\right] \approx 0$$

Where $\mathcal{O}$ stands for the oracle access to $\mathcal{A}$ pursuing the interactions.

Definition 8 (Succinctness, SNARK). *A non-interactive argument system $\mathcal{X}$ for a relation $\mathcal{R}_\lambda$ is **succinct** if the proof π produced by the prover has size $o(|w|)$ and the run-time of the verifier is $o(|w|)$ for all relations $\mathcal{R}$ drawn from $\mathcal{R}_\lambda$. A non-interactive argument system with this property is called SNARK.*

2.3 Polynomial commitment

We conveniently adapt the definition of polynomial commitment given by [2, 7] (to its non-interactive version) to match the formalism of the present document. Formally, a polynomial commitment is a tuple of p.p.t. algorithms (Setup, Commit, Prove, Verify) where,

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, t)$ generates the public parameters pp suitable to commit to polynomials of degree $< t$. It is to be done by a trusted authority.
- $C \leftarrow \mathsf{Commit}(\mathsf{pp}, P(X))$ outputs a commitment C to a polynomial $P(X)$ of degree at most t using pp.
- $(x, y, \pi_x) \leftarrow \mathsf{Prove}(\mathsf{pp}, P(X), x)$ outputs (x, y, π_x) where π_x is a proof for the evaluation of $y = P(x)$.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{pp}, C, x, y, \pi_x)$ verifies that $y = P(x)$ is the correct evaluation of the polynomial committed in C.

The correctness and security of Polynomial Commitments Schemes are defined in Supplement A. We use the KZG polynomial commitment scheme [25] to informally refer to the polynomial commitment based on bilinear groups assumptions (Supplement D). This protocol has been widely studied, extended and applied in numerous recent works [2], [17], [14].

2.4 Fiat-Shamir

Informally, the Fiat-Shamir heuristic is a tool that allows transforming interactive protocol from a specific class into non-interactive protocols. This specific class is known as public-coin protocols and is formally defined as follows.

Definition 9 (Public Coin). *An interactive protocol between a prover and a verifier is* ***public-coin*** *if all the messages sent by the verifier to the prover are randomly and independently sampled from the messages sent by the prover (that is, random coins from the verifier are publicly available).*

In Supplement J, we provide more details on how we adapt and instantiate the Fiat-Shamir heuristic for our aim in Supplement J and Supplement E.

2.5 MiMC

In this section and in Section 3, we describe the structure of MiMC and the idea of GKR, the reader familiar with these concepts can skip them and directly go to Section 3.1.

We summarize the construction of MiMC [1]. Let q be a prime, and $\mathbb{F}_q$ be the finite field of order q. Let α be the smallest integer co-prime with $q-1$; the map $x \to x^\alpha$ defines a bijection of $\mathbb{F}$. Let $(c_i)_{0 \leq i < d}$ be a sequence of $\mathbb{F}$. Define the round function $R_i(x,k) = (x+k+c_i)^\alpha$. The MiMC secure pseudo-random permutation is defined as the composite $\mathit{\Pi}(x,k) = R_{d-1}(\bullet,k) \circ R_{d-2}(\bullet,k) \circ \ldots \circ R_1(\bullet,k) \circ R_0(x,k)$. The authors of [1] suggest that d be chosen so that $d \geq \left\lceil \frac{\log(q)}{\log \alpha} \right\rceil$ and that the round constants (c_i) be drawn independently and uniformly at random in $\mathbb{F}_q$. The MiMC symmetric block cipher is obtained by $\mathsf{Enc}(x,k) = \mathit{\Pi}(x,k) + k$. From the cipher, we can obtain a hash function using the Miyaguchi-Preneel construct [29, 6].

2.6 Sumcheck protocol

The sumcheck protocol [27] is a multi-round interactive protocol for the following relation where P and a are public inputs and the witness is empty.

$$\left\{ (P(X), a;\ ) : \sum_{x_{k-1} \in \{0,1\}} \cdots \sum_{x_0 \in \{0,1\}} P(x_0, \ldots, x_{k-1}) = a \right\}$$

where P is a k multivariate polynomial of maximal degree d on each variable. It consists of k rounds of complexity $O(d)$, each doing sensibly the same thing (from the verifier's point of view) and a special final round where the verifier performs an evaluation of $P(X)$ at a random challenge point and compares the result with the prover's messages.

Remark 1. Giraffe describes how to obtain a prover with time linear in the number of gates in the circuit. This is applicable only if the sumcheck has a specific form. Libra [34] subsequently proposed an approach for the more general case where the circuit does not necessarily have a data-parallel structure.

Remark 2. A useful takeaway to understand the role the sumcheck plays in GKR is to notice that the sumcheck protocol reduces a claim about an exponential size sum of values of P to a claim on a single evaluation of P at a random point.

3 GKR Protocol; A public-coin argument system

Description As mentioned, GKR generates proofs for the data-parallel execution of a layered arithmetic circuit. At a high level, it is done by iteratively applying a sequence of sumcheck protocols [27], one for each layer of the circuit. Each iteration of the sumcheck protocol inside GKR establishes consistency between two successive layers of computation (starting with the output layer, layer 0, and working backwards to the input layer, layer d). While the current section introduces the relevant definitions and details our contributions relative to the GKR protocol itself, the reader can find a description of the protocol in Supplement B.

Each run of the sumcheck protocol *a priori* requires the verifier to perform a costly polynomial interpolation (the special last round). GKR bypasses this completely: the prover provides the (supposed) value of that interpolation, and another instance of the sumcheck protocol is invoked to prove the correctness of this value. In GKR, the only time the final round of the sumcheck protocol is performed by the verifier is at the last layer (layer d: input layer). Here, we give the formal description of GKR for a more general circuit-family as in [35].

In the following section, we will often use the following notations. Let $\mathcal{G}$ be a directed acyclic graph. $E(\mathcal{G})$ denotes the set of edges of $\mathcal{G}$ and $V(\mathcal{G})$ the vertex. For two vertices v, v', (v, v') is the edge going from v to v'. We assume a total ordering over $V(\mathcal{G})$ compatible with the natural partial ordering defined by the edges of $\mathcal{G}$. $I : V \to V^*$ maps each vertex v to the ordered list of vertices $I(v) = \{v' \in V(\mathcal{G}) : (v', v) \in E(\mathcal{G})\}$ such that $\forall v' \in I, (v', v) \in E(\mathcal{G})$. Conversely, $O(v) = \{v' \in V(\mathcal{G}) : (v, v') \in E(\mathcal{G})\}$. We will use directed acyclic graphs to describe the "shape" of a computation, as such we call any vertex v with $I(v) = \emptyset$ an **input gate**, and any vertex v with $O(v) = \emptyset$ an **output gate**. In the following, eq denotes the multilinear polynomial $\mathsf{eq}(X_{0\cdots n-1}, Y_{0\cdots n-1}) = \prod_{i<n} [X_i Y_i + (1 - X_i)(1 - Y_i)]$.

Layered arithmetic circuits In the following, we give a broad definition of an arithmetic circuit that encompasses gates with arbitrary low-degree multivariate polynomials (as opposed to just additions and multiplications). This broad definition will be useful for specifying custom gates as in [18] (e.g., in Section 3.1, Supplement C for applying GKR over the MiMC and Poseidon permutations).

Definition 10. *An **arithmetic circuit** $\mathcal{C}$ over a ring $\mathbb{K}$ is a pair $(\mathcal{G}, f)$ where $\mathcal{G}$ is a directed acyclic graph and f maps vertices v of G to $|I(v)|$-multivariate polynomials over $\mathbb{K}$.*

Definition 11 (batch assignment). *For $n \in \mathbb{N}$, a **batch assignment** $\mathcal{B}$ for an arithmetic circuit $\mathcal{C} = (\mathcal{G}, f)$ on $\mathbb{K}$ is a mapping from $V(\mathcal{G})$ to the set of n-multilinear polynomials such that*

$$\forall x \in \{0,1\}^n, \forall v \in V(\mathcal{G}), \quad \mathcal{B}(v)(x) = R_v(\mathcal{B}(u_0)(x), \ldots, \mathcal{B}(u_{k-1})(x))$$

where we note $I(v) = \{u_0, ..., u_{k-1}\}$ and $R_v = f(v)$. Equivalently, this corresponds to 2^n assignments of an arithmetic circuit. We recall that

In other words, $\mathcal{B}(v)$ interpolates $R_v(\mathcal{B}(u_0), \ldots, \mathcal{B}(u_{k-1}))$ on the hypercube $\{0,1\}^n$. If v is an input gate or an output gate, we call $\mathcal{B}(v)$ (resp.) an input or output polynomial.

Remark 3. If $\mathcal{B}$ is a batch assignment to $\mathcal{C}$, then for all $v \in V(\mathcal{G})$, if we note $I(v) = (u_0, ..., u_{k-1})$ and $P = f(v)$, then

$$\forall x \in \mathbb{K}^n, \mathcal{B}(v)(x) = \sum_{h \in \{0,1\}} \mathsf{eq}(x, h) P(\mathcal{B}(u_0)(h), \ldots, \mathcal{B}(u_{k-1})(h))$$

The later remark is fundamental to the GKR protocol. It gives a relation between a vertex assignment and its input layer in the form of a summation.

Fiat-Shamir transform The GKR protocol is not constant-round and thus, one cannot straightforwardly compiles it in the random oracle model. Recent works [15, 4, 12] have studied the Fiat-Shamir transform of the GKR protocol and the Fiat-Shamir of the sumcheck protocol. We give an informal justification for the soundness of GKR in the random oracle model below.

In [12] Canetti et al. proved that GKR is round-by-round sound in the so-far transcript model. [7] Additionally, the authors of [12] argue that *round-by-round soundness* readily implies security against state-restoration attack, which is a notion introduced in [4]. In [4] the authors argue that if a public-coin interactive protocol is secure against state restoration attack, then its non-interactive version via Fiat-Shamir is also sound in the random oracle model. Putting it together, this means GKR is sound in the (so-far transcript) random oracle model. We also highlight that GKR is widely used in the random oracle model [37, 34, 11].

In our compiler, we essentially need a single-round version of GKR. In essence, only the first round of communication is initially kept interactive, while all the other rounds are compiled using the Fiat-Shamir. We elaborate on this version, its security and instantiation in Supplements E and J.

When embedded in R1CS (or more broadly, any type of algebraic circuit that does not have special support for hash functions), the performance of the verifier is dominated by the Fiat-Shamir hashes. Particularly, the verifier generates the first randomness by hashing the statement supposed to be proven by GKR, this incurs the prover to perform Fiat-Shamir hashes of size $(|x|+|y|)$ (corresponding to the claim $H_{mimc}(x) = y$). Since doing so entails the verifier to work with more computation than it would need to perform the hash itself. We treat this issue in Section 5.4. Additionally, the verifier and prover perform a logarithmic number of hashes due to applying Fiat-Shamir to the sumchecks instances.

[7] The authors of [12] also argues that their transformation is sound in the standard model. We do not use this fact in this work.

Performances The verifier work consists of evaluating once each of the input and output polynomials. It also has the overhead of the sumchecks, one for each vertex with a single fan-out and two for the layers that need a multi-fan out (because it also needs to run protocol 2 (described in Supplement B). Each of the sumchecks has a logarithmic runtime for the verifier. On its hand, the prover runtime is driven by the sumchecks runtime. They have a runtime of $O(N)$ for N the number of instances of the batch assignment.

Compiling the GKR verifier in a R1CS The evaluation of the input and output polynomials requires few multiplication gates: 1 for each input and 1 for each output. Sumchecks are, however, expensive to verify in practice, and their cost is driven by the hashes required by the Fiat-Shamir transform. Even though this is a logarithmic overhead, it has large constants and typically occupies between 1M to 10M constraints (with Groth16) and twice more with Plonk, depending on the number of hashes to be proven in the batch.

3.1 Custom gates for the GKR protocols

This section presents our contributions in using the GKR protocol for the MiMC keyed permutation. We additionally describe a set of custom gates for the Poseidon permutation in Supplement C. Following a suggestion in Libra [34], we define a custom family of gates. Let α be the exponent for MiMC in $\mathbb{F}$, and d be the number of rounds for the MiMC permutation. Recalling Section 2.5, with $c_1, \ldots, c_d$ being elements in $\mathbb{F}$, we define $R_{c_i,\alpha}$ as the polynomial:

$$R_{c_i,\alpha}(X_0, X_1) = (X_0 + X_1 + c_i)^\alpha$$

We then give an overview of the arithmetic circuit $\mathcal{C}$ for the MiMC permutation in figure Fig. 1.

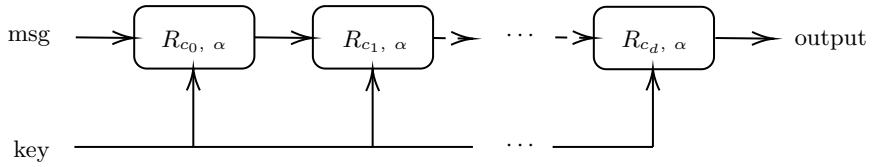

Fig. 1. Structure of the arithmetic circuit for MiMC

As highlighted in Fig. 1, the gate $R_{c_i,\alpha}(X_0, X_1)$ can just be sequentially repeated with the appropriate constants and number of rounds to instantiate a variant of the GKR protocol specialized for the MiMC permutation. In Supplement C, we present the custom gates for the Poseidon hash.

4 Challenge of Recursion over Public-Coin Argument Systems

In this section, we explain the challenge of recursion over public-coin argument systems.

4.1 Concrete Example

Before delving into the abstract matter, we first give a concrete example of how we intend to embed GKR in a SNARK. Consider, for instance, the problem of verifying Merkle proofs in, say, Groth16 [23]. The circuit doing this performs two distinct tasks: (1) routing Merkle paths (i.e., preparing the inputs to be hashed and deciding the order in which to hash them) and comparing the final output hash with a public Merkle root hash, (2) computing actual hashes.

The circuit performing the first task, call it $\mathcal{C}'$, simply believes the values output by the hashing sub-circuit. The most straightforward option for the second task is to implement the hash as a sub-circuit, i.e., do the second task by *computing* the hashes in the circuit. Our solution is different: it is to *verify* the requisite hashes using a GKR verifier sub-circuit. Thus, combining $\mathcal{C}'$ and the GKR verifier circuit produces a circuit $\mathcal{C}$ that verifies Merkle proofs in their entirety.

Naive attempt We could try to use the GKR verifier inside $\mathcal{C}$ in its original form (non-interactive by Fiat-Shamir). The Fiat-Shamir transform of the GKR protocol has been well studied, and we know it is sound for rightfully chosen hash functions. The result would be a sound protocol; however, this approach comes with a major impediment. In the interactive version, the verifier is asked to send a challenge to the prover after receiving the response from the prover. In the Fiat-Shamir transform of GKR, this implies that the verifier has to hash all the information it has received so far, including the entire GKR statement: 3 times worse than directly checking the hashes in an arithmetic circuit. Our goal here is to circumvent the burden of hashing the entire GKR statement in order to verify a (non-interactive) GKR proof inside the circuit.

Note that in the Fiat-Shamir transform version that we use, the new randomness (for the new round) is obtained by hashing the previous randomness and the last message, ignoring the past transcript (see Supplement J). Thus, the bottleneck in applying GKR for hash verification is generating a challenge for the first round. That is why we refer to the challenge for the first round as the "Initial Randomness", and we consider a public-coin *single-round* interactive argument system rather than a multi-round one.

5 Our Compiler for Recursion

This section introduces a generic compiler for building a special class of recursion systems. Let the argument system $\mathcal{X}_b$ be a public-coin single-round interactive

argument system (corresponding to GKR in our use-case). Our compiler aims at combining $\mathcal{X}_b$ with SNARK system $\mathcal{X}_a$ to get an efficient recursion system (running the verifier of $\mathcal{X}_b$ inside $\mathcal{X}_a$). Based on the challenge we described in Section 4 and with the help an extra proof-system for knowledge of committed value $\mathcal{X}_\mathcal{T}$ – which we specify in the later in the present section – we develop our compiler in such a way that it can solve the problem of initial randomness. Here we give a general intuition of how to build such compiler.

5.1 Intuition

The compiler goes through two main steps, in the first step we assume that $\mathcal{X}_b$ is a single-round argument system (where the first prover's message is not included in the circuit) and the challenge is available as part of the public input, in the second step we replace the first message of the prover with a short commitment (to the public inputs). A more detailed intuition is given in the following, where we still stick to our use-case; embedding GKR inside Groth16 for Merkle proofs.

Protocol 1 We embed the GKR verifier into a sub-circuit of $\mathcal{C}$ alongside $\mathcal{C}'$. We use the GKR statement and the initial randomness as a public input of the embedding circuit. The resulting circuit $\mathcal{C}$ can be succinctly described as in Fig. 2. While Fig. 3 describe the steps of the protocol. Note that the circuit involved in the recursion is the part after receiving the challenge ρ (in Fig. 3).

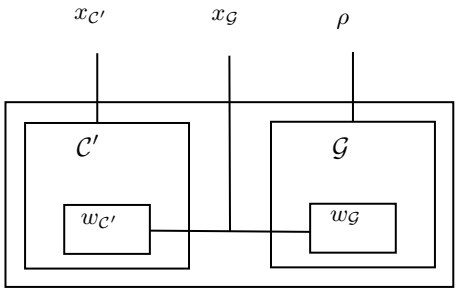

Fig. 2. Protocol 1 : Circuit $\mathcal{C}$ construction. $x_{\mathcal{C}'}$ and $w_{\mathcal{C}'}$ are the public input vector and witness vectors of $\mathcal{C}'$. $\mathcal{G}$ is a circuit embedding the one-round GKR verifier. The GKR proof belongs to $w_\mathcal{G}$ (witness of $\mathcal{G}$). $x_\mathcal{G}$ is the GKR statement vector, and ρ is the initial randomness.

Notice the following two points. First, Protocol 1 is *interactive* (public-coin single-round), as such it is *not* a SNARK. Secondly, the public input vector is longer (it now contains the GKR inputs/outputs). This is highly undesirable: the verifier is no-longer sublinear in the circuit size. This is, for instance, a problem when verifying a Merkle proof. In this case, the protocol loses its succinctness.

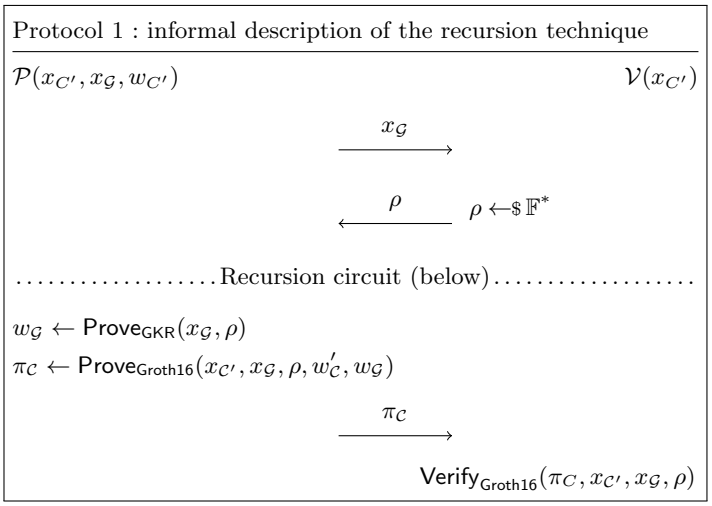

Fig. 3. Protocol 1, We omit the passing of the public parameters. The meaning of the variable is the one of Fig. 2

It sounds like a step backward. Nonetheless, one can argue that this protocol is sound. This will be helpful for analyzing the Protocol 2.

Protocol 2 Protocol 2 improves on Protocol 1 by removing the unnecessary GKR inputs/outputs from the public inputs of the circuit. This greatly reduces the verifier's overhead and brings back the succinctness that we lost with protocol 1. We start from an observation in the inner-working of the Groth16 presented in Fig. 4. Crucially, the output of the pairing check can be computed with only access to γ. Secondly, doing a multi-scalar multiplication (MSM) of a vector of field element and a set of group elements for which no discrete log is known can be viewed as a binding commitment analog of the Pedersen commitment.

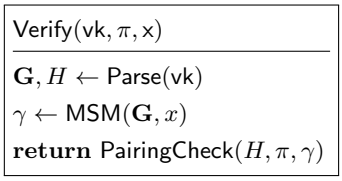

Fig. 4. Simplified take on the Groth16 verifier

Note that in verification of Groth16 (Fig. 4), we have an MSM where the entries of $\mathbf{G} = \mathbf{G}_{\mathcal{C}'} \| \mathbf{G}_{\mathcal{G}} \| \mathbf{G}_{\rho}$ correspond to the entries of $x_{\mathcal{C}'}, x_{\mathcal{G}}, \rho$ in the MSM. The idea of protocol 2, is that instead of sending the public inputs to the verifier, the prover computes $\gamma_{\mathcal{G}} = \mathsf{MSM}(\mathbf{G}_{\mathcal{G}}, x_{\mathcal{G}})$ and sends it to the verifier. From there,

the protocol continues as in Protocol 1. When he checks the Groth16 proof, the verifier chooses ρ and completes MSM by adding the missing parts to $\gamma_{\mathcal{G}}$.

An issue, is that just doing that is insecure. Indeed, a malicious prover can pass the verification check for any arbitrary (invalid) $x'_{\mathcal{C}'}$ by sending $\gamma'_{\mathcal{G}} = \gamma_{\mathcal{G}} + \mathsf{MSM}(\mathbf{G}_{\mathcal{C}'}, x_{\mathcal{C}'} - x'_{\mathcal{C}'})$ where $x_{\mathcal{C}'}$ belongs to $\mathcal{L}(\mathcal{C}')$ for which the prover knows a witness (we shall call such attack as mix-and-match).

We rule out this attack by additionally requesting the prover to send an argument of knowledge that she knows $x_{\mathcal{G}}$ such that w.r.t $\mathbf{G}_{\mathcal{G}}$ we have $\gamma_{\mathcal{G}} = \mathsf{MSM}(\mathbf{G}_{\mathcal{G}}, x_{\mathcal{G}})$. This ensures that the prover cannot use anything aside group elements in $\mathbf{G}_{\mathcal{G}}$ in the claim of $\gamma_{\mathcal{G}}$. The Protocol 2 is sound if Protocol 1 is sound (that implies the binding property of $\gamma_{\mathcal{G}}$) and if the argument of knowledge ensuring the right computation of $\gamma_{\mathcal{G}}$ is sound as well.

Finally, we apply the Fiat-Shamir transform, where the initial randomness is computed as $\rho = H_{\mathcal{FS}}(\gamma_{\mathcal{G}})$. This removes the only interaction of the protocol. Note that we no longer require the verifier to hash the GKR statement, but rather its commitment $\gamma_{\mathcal{G}}$ which is only one group element.

5.2 Preliminaries for our compiler

Our compiler has three layers. The first layer verifies the GKR proof, or rather, all the parts that come after the initial randomness, inside a SNARK ($\mathcal{X}_a$) and sets all the public inputs of GKR, i.e., $x_{\mathcal{G}}$ as public inputs of the resulting SNARK. The second layer of compilation assume that the SNARK $\mathcal{X}_a$ has a set of properties that we formalize in the present section (2-Step verification with splitting compatibility for the "computation" step) and specify in Section 5.3. This property relates to how the verifier handles the public input. Informally, we require that computation relative to the public inputs, the *Computation step*, can be factored out of the rest of the verifier's computation, the *Justification step*. This must be possible in such a way that the public inputs do not appear in the *Justification step*, only γ, the result of the "Computation step". We formalize this as **2-steps verification** Finally, we require that the *Computation step* can be computed by "recombining" two partial intermediate results obtained from two complementary subsets of the public input. The latter property is what we formalize as **splitting-compatibility**. The last layer just applies the Fiat-Shamir transform.

Definition 12 (2-Steps Verification). *We say that a SNARK system has 2-step verification if the verification algorithm can be expressed as follows,*

$$\mathsf{Verify}(\mathsf{vk}, x, \pi) = \begin{cases} 1.\ \textit{Computation: } c \leftarrow F(\mathsf{vk}, x, \pi) \\ 2.\ \textit{Justification: } 0/1 \leftarrow \mathsf{Verify}'(\mathsf{vk}, c, \pi)). \end{cases}$$

Where the input x is not used in the justification-step and c is much shorter than x (i.e., F is compressing, note that this requirement implies non-trivial choice of F). We emphasize that the computation-step may itself include several steps of computations.

Remark 4. The 2-steps verification property becomes interesting (and less trivial) when we impose a special property on the computation-step called splitting-compatible, which we explain in the following.

Example 1. The verifier of Groth16 [23] satisfies the above property. Loosely speaking, the first part consists in performing a MSM of the public inputs with a subset of the verification key see the Fig. 5 and the second part is the pairing check. For PLONK, identifying the *computation step* is non-trivial. We elaborate on it in Section 6.2. At a high-level, it amounts to computing two things : a challenge $\mathfrak{z}$ and evaluating a polynomial $\mathsf{PI}(\mathfrak{z})$ interpolating the public inputs vector.

We recall the pairing equation of the Groth16 verifier below, with the notations of [23] below

$$[A]_1[B]_2 = [\alpha]_1[\beta]_2 + (\sum_{i=0}^{n} ai[\frac{\beta ui(x) + \alpha vi(x) + wi(x)}{\gamma}]_1).[\gamma]2 + [C]_1[\delta]_2$$

Fig. 5. Groth16's verifier equation

We now, introduce the notions of *splitting* and *partitioning*. Informally, they can be understood as dividing a string to two sub-strings (Fig. 6).

Definition 13 (Splitting, Partitioning). *We define splitting (res. partitioning) as the map $\phi : X \to X_A \times X_B$ that maps a vector X to two sub-vectors X_A, X_B as follows. Let I be the index set associated with entries of the vector X, i.e., $I = 1, \ldots, |X|$. We denote a sub-vector of X associated with the index-set $A \subseteq I$ as X_A (using the indexes of A in ascending order).*

We say that X_A, X_B is a splitting of X if for $A, B \subseteq I$ we have $A \cup B = I$. It is partitioning if A, B are partitioning of I as well, namely, $A \cap B = \emptyset$. A visualization is given in Fig. 6. For convenience, we may abuse the notation to say A, B is a splitting (or partitioning) of X as $X = (A, B)$.

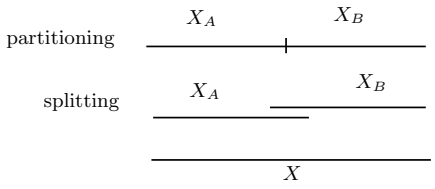

Fig. 6. Splitting and partitioning of X to X_A, X_B.

A splitting-compatible map f is a map that can be split and recombined according to a splitting σ of its inputs (Fig. 7). As a toy example, one can

consider the Pedersen commitment $g_1^{a_1} \cdot g_2^{a_2}$ splitting to $g_1^{a_1}$ and $g_2^{a_2}$ according to the splitting (a_1, a_2) to a_1 and a_2. The formal definition is as follows.

Definition 14. *(σ-Compatibility) Consider finite subsets $A, B, X, Y \subset \{0,1\}^*$, a map $f : X \to Y$, and a partitioning over its input space as $\sigma : X \to A \times B$. We say that f is σ-**compatible** if there exists $f_A : A \to Y_A$ and $f_B : B \to Y_B$ and a combiner $g : Y_A \times Y_B \to Y$ such that $\forall (a,b) \in A \times B, f(\sigma^{-1}(a,b)) = g(f_A(a), f_B(b))$ (see Fig. 7). We may call f_A, f_B as the splitting of the map f.*

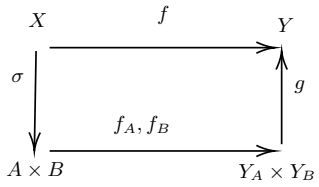

Fig. 7. Splitting-Compatible: the map f is compatible with the splitting σ.

5.3 Building Blocks of The Compiler

The compiler we describe in the following subsection works with several argument systems $(\mathcal{X}_a, \mathcal{X}_b, \mathcal{X}_\mathcal{T})$ as input that we properly introduce later in the current section. They must satisfy several requirements that we also specify below.

Requirements on $\mathcal{X}_a$:

- $\mathcal{X}_a = (\mathsf{Setup}_a, \mathsf{Prove}_a, \mathsf{Verify}_a)$ should be a secure SNARK scheme for a family of relations $\mathcal{R}_{\mathcal{F}0}$ closed under intersection.
- The verification algorithm Verify_a is a 2-step verification (see Definition 12).
- Let F_a be an algorithm in the computation-step of the verification (cf Definition 12), then for any vk and any partitioning σ of x, the function $F_a(\mathsf{vk}, \bullet)$ should be compatible with σ. In the rest of the paper, we may call such F_a or its output as the contribution of the public input. We may also use $F_a(x)$ when vk is implicit. We emphasize that, if computation-step includes several computations, each of these computations should be splittable (i.e., σ-compatible).

Remark 5. Note that the soundness of $\mathcal{X}_a$ implies that (Setup_a, F_a) — viewed as a commitment scheme of the public inputs — must be binding as well.

Requirement on $\mathcal{X}_b$:

We require $\mathcal{X}_b = (\mathsf{Setup}_b, \mathsf{Prove}_b, \mathsf{Verify}_b)$ being a public-coin single-round interactive argument of knowledge for some relation family $\mathcal{R}_{\mathcal{F}1}$ without any specific

restriction on $\mathcal{R}_{\mathcal{F}1}$. Furthermore, as the protocol is single-round of interaction Verify_b (and also Prove_b) can be split in two part. Each part, $\mathsf{Verify}_{b,1}, \mathsf{Verify}_{b,2}$ executes respectively the first and the second round of the verifier. We use the notation $\pi_{b,1}, \pi_{b,2}$ for the prover messages at round (resp.) 1 and 2. Finally, we require that $\mathsf{Verify}_{b,2}$ be expressible in a $\mathsf{poly}(\lambda)$-sized instance of $\mathcal{R}_{\mathcal{F}1}$.

Remark 6. Implicitly, we want to use the GKR protocol as $\mathcal{X}_b$. An apparent impediment is that GKR is not a single-round protocol, as required. We address it in Supplement E, where we compile the GKR (non-interactive version) into a single-round protocol. A second apparent issue is that in practice, GKR does not prove or argue knowledge of a witness. This corresponds to the trivial case where the witness is an empty string.

Requirement on $\mathcal{X}_\mathcal{T}$:

We introduced above F_a being defined in the computation-step of verification of $\mathsf{Verifier}_a$ and σ being a partitioning of the public inputs. Recall that we already required that F_a be compatible for all partitioning $\sigma : X \to L \times R$ (where $\mathcal{X}$ is the set of public input of $\mathcal{X}_a$). As a consequence, we can introduce splittings of the map F_a as ϕ_L, ϕ_R and its combiner g (see Definition 14).

Note that in our protocol, we split the public input x to (x_L, x_R), and delegate the computation of the map ϕ_L to the prover. We require $\mathcal{X}_\mathcal{T}$ being a succinct non-interactive argument of knowledge for relations of the form $\mathcal{R}_\mathcal{T}(\mathsf{vk}, \sigma) = \{(\gamma; x_L) : \gamma \leftarrow \phi_L(x_L)\}$ where vk is the verification key of $\mathcal{X}_a$.

Remark 7. $\mathcal{X}_\mathcal{T}$ works over a relation that is defined for fixed vk and σ that are, respectively, the verification key of $\mathcal{X}_a$ and the splitting over the public inputs of $\mathcal{X}_a$. This point is crucial as it addresses the mix-and-match attack raised in Section 5.1. For the sake of clarity, the role of $\mathcal{X}_\mathcal{T}$ is not just about extracting x_L s.t. $\gamma = \phi_L(x_L)$ but to enforce that γ was obtained only using the correct ϕ_L and no other information.

Remark 8. Following the Remark 5, $\phi_L(\cdot)$ (and similarly for ϕ_R) can be seen as a binding commitment scheme as well. Indeed, if an adversary were able to find $x_L \neq x'_L$ such that $\phi_L(x_L) = \phi_L(x'_L)$, then for all x_R, we have $x = \sigma^{-1}(x_L, x_R) \neq x' = \sigma^{-1}(x'_L, x_R)$ and $F_a(\mathsf{vk}, x) = F_a(\mathsf{vk}, x')$ which contradicts the soundness of $\mathcal{X}_a$. The binding property is interesting to intuitively see that we can replace x_L with $\gamma_v = \phi_L(x_L)$ in the Fiat-Shamir transform, though in our security proofs we directly reduce the security to the knowledge-soundness.

5.4 Formal Description of Compiler

The compiler proceeds in three layers of compilation, see Fig. 8. Intuitively, the first step consists in recursing the verifier of $\mathcal{X}_b$ (more precisely, the part coming after the prover response to the random challenge) inside the proof $\mathcal{X}_a$, where $\mathcal{X}_b$ additionally passes its public inputs as part of the public inputs of the SNARK scheme $\mathcal{X}_a$. Then, a second layer of compilation allows us to delegate

some contributions of public-inputs of $\mathcal{X}_b$ (e.g., a subpart of MSM in the contexts of Groth16 [23]) from the verifier's computations to the prover's. The last layer simply consists in applying the Fiat-Shamir transform.

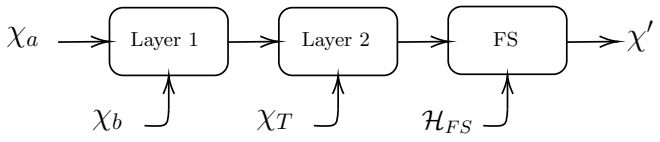

Fig. 8. Overview of the compiler

Now we are ready to present the inner-work of each layer separately, as we are going through the layers we prove the security for each layer.

The first layer The first layer (L1) takes as input a SNARK scheme $\mathcal{X}_a$ for a relation $\mathcal{R}' \in \mathcal{R}_{\mathcal{F}0}$, and a public-coin single-round argument of knowledge $\mathcal{X}_b$ for a relation, $\mathcal{R}_1 \in \mathcal{R}_{\mathcal{F}_1}$ as in Section 5.3. Note that since $\mathcal{X}_b$ is recursed inside $\mathcal{X}_a$, then $\mathcal{X}_a$ should also check for some relation between the public input and witnesses $\mathcal{R}_1$ and the rest of the circuit. Recalling the visualization given in Fig. 2, here one can imagine $\mathcal{R}_0$ as the circuit $\mathcal{C}'$, $\mathcal{R}_1$ for the GKR, and the connection between $\mathcal{R}_0$ and $\mathcal{R}_1$ is checked through some equality relations in the form of splittings). Thus, the relation for $\mathcal{X}_a$ can be expressed as follows,

$$\mathcal{R}' = \left\{ (x, \rho, \pi_{b,1}; w) : \begin{array}{c} x_0, x_1 = \sigma_t(x),\ w_0, \pi_{b,2} = \sigma_u(w) \\ 1 \leftarrow \mathsf{Verify}_b(\mathsf{pp}_1, x_1, \pi_{b,2}, \rho) \\ 1 \leftarrow \mathcal{R}_0(x_0, w_0) \end{array} \right\} \tag{1}$$

Where the compiler also receives two splittings σ_t, σ_u of the public input and the witness spaces. Moreover, we require σ_u being **partitioning** but not σ_t. In Supplement F we discuss the choice of σ_t, σ_u.

Intuitively, the first layer inherits the security property of the underlying SNARK $\mathcal{X}_a$ and argument system $\mathcal{X}_b$ and so for any choice of σ_u, σ_t yields a secure protocol for the following relation, which is also depicted in Fig. 9.

$$\mathcal{R}_{L1}(\mathcal{R}_0, \mathcal{R}_1, \sigma_t, \sigma_u) = \left\{ (x \in \mathsf{X}; w \in \mathsf{W}) : \begin{array}{cc} x_0, x_1 = \sigma_t(x), & \mathcal{R}_0(x_0, w_0) = 1 \\ w_0, w_1 = \sigma_u(w), & \mathcal{R}_1(x_1, w_1) = 1 \end{array} \right\} \tag{2}$$

Remark 9. This way of combining relations only allows the two instances $\mathcal{R}_0$ and $\mathcal{R}_1$ to share some part of their public inputs, but not their witnesses.

In our use case (SNARK over GKR, for Merkle tree), some public input of $\mathcal{X}_a$ are used in the Merkle tree (e.g., because we need to commit to them). This is why x_1 as the public-input of $\mathcal{R}_1$ may share some entries with x_0, the public-input of $\mathcal{R}_0$. More in details, σ_t not being a partitioning (but only a simple

splitting) implies equality constraints in addition to the constraints specified by $\mathcal{R}_0$ and $\mathcal{R}_1$.

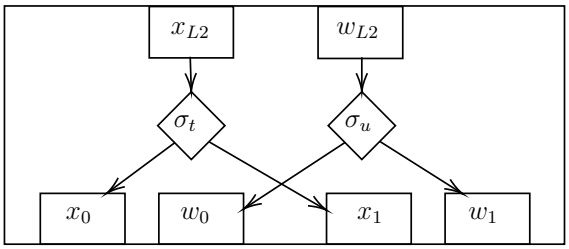

Fig. 9. Representation of the layer 1 compiler

Layer 1 Construction. Fig. 10 describes the construction of layer 1 with inputs $\mathcal{R}_0, \mathcal{R}_1, \sigma_t, \sigma_u, \mathcal{X}_a, \mathcal{X}_b$

$\underline{\text{Setup}(1^\lambda, \mathcal{R}_{L1})}$

$\mathcal{R}_0, \mathcal{R}_1, \sigma_t, \sigma_u \leftarrow \mathsf{Parse}(\mathcal{R}_{L1})$

$\mathsf{pp}_1 \leftarrow \mathsf{Setup}_1(1^\lambda, \mathcal{R}_1)$

$\mathsf{pp}' \leftarrow \mathsf{Setup}_a(1^\lambda, \mathcal{R}')$ for relation $\mathcal{R}'$ in Eq. (1)

return $\mathsf{pp} = (\mathsf{pp}_1, \mathsf{pp}')$

Interactive Argument of Knowledge

$\underline{\mathsf{Prove}_{L1}(\mathsf{pp}, x, w_0, w_1)}$		$\underline{\mathsf{Verify}_{L1}(\mathsf{pp}, x)}$
$\mathsf{pp}_1, \mathsf{pp}' \leftarrow \mathsf{Parse}(\mathsf{pp})$		
$\pi_{b,1} \leftarrow \mathsf{Prove}_1(\mathsf{pp}_1, x_1, w_1)$		
	$\xrightarrow{\pi_{b,1}}$	
	$\xleftarrow{\rho}$	$\rho \leftarrow\$ \mathcal{C}$
$\pi_{b,2} \leftarrow \mathsf{Prove}_1(\mathsf{pp}_1, x_1, \rho, w_1)$		
$w' \leftarrow (w_0, \pi_{b,1}, \pi_{b,2})$		
$\pi' \leftarrow \mathsf{Prove}_0(\mathsf{pp}', (x, \rho), w')$		
	$\xrightarrow{\pi'}$	$\mathsf{Verify}_0(\mathsf{pp}', (x, \rho), \pi')$

Fig. 10. Setup and interactions of layer 1

Remark 10. here we are using an equivalent representation of Fig. 3 where the verifier has all the public input of $\mathcal{X}_b$ as its input. This representation is more compatible with the definition of SNARK, where the prover and the verifier receive the same public input.

Completeness: It follows from the completeness of $\mathcal{X}_a$ and $\mathcal{X}_b$. More precisely, the completeness of $\mathcal{X}_b$ guarantees that for the correct statement $(x_1, \rho) \in \mathcal{L}_{\mathcal{R}_1}$ where $\mathcal{R}_1(x_1, \rho; w_1) = 1$, the verification of argument system $\mathcal{X}_b$ satisfies $\mathsf{Verify}_1(\mathsf{pp}_1, x_1, \rho, w_1) = 1$. Which gives the right relation consumed by the argument system $\mathcal{X}_a$, then the completeness of $\mathcal{X}_a$ implies that verification algorithm of $\mathcal{X}_a$ (which is also the verification associated with $\mathcal{R}_{L1}$) outputs 1.

Knowledge-Soundness. Here, we denote our argument system for the layer 1 as $\mathcal{X}_{L1}$ applied over the corresponding relation $\mathcal{R}_{L1}$ (in Eq. (2)). Let $\mathcal{E}_a$ and $\mathcal{E}_b$ be the extractors respectively associated with the argument systems $\mathcal{X}_a$ and $\mathcal{X}_b$. If $\mathcal{X}_{L1}$ outputs a valid proof, we can run the extractor $\mathcal{E}_a$ to extract a witness w including two main parts; a witness for $\mathcal{R}'$ (relation associated with $\mathcal{X}_a$) and a proof associated with $\mathcal{X}_b$. There are two possible cases;

- **Case 1.** The witness for $\mathcal{R}'$ is correct (i.e., satisfies the relation $\mathcal{R}'$) : this means verification $\mathcal{X}_b$ passed, and by knowledge-soundness of $\mathcal{X}_b$, the probability that the extractor $\mathcal{E}_b$ fails (i.e., it extracts a non-satisfying witness for $\mathcal{R}_1$ from the proof of $\mathcal{X}_b$) is negligible.
- **Case 2.** The relation $\mathcal{R}'$ is not satisfied : it means the extractor $\mathcal{E}_a$ failed, but it can only happen with negligible probability by knowledge-soundness of $\mathcal{X}_a$.

Wrapping up everything together, the extractor $\mathcal{E}_a$ outputs the correct witness (w_0, π) with overwhelming probability. This give a valid proof π for $\mathcal{X}_b$ (as part of the extracted witness), which then the extractor $\mathcal{E}_b$ can use it to extract the correct witness w_1.

Theorem 1 (Knowledge-Soundness of $\mathcal{X}_{L1}$). *Let $\mathcal{X}_a, \mathcal{X}_b$ be as required in Section 5.3. If $\mathcal{X}_a$ and $\mathcal{X}_b$ have (resp.) knowledge errors ϵ_a and ϵ_b. Then, the aforementioned protocol $\mathcal{X}_{L1}$ has knowledge error $\epsilon = O(\epsilon_a + \epsilon_b)$.*

Proof-Sketch. Let $\mathcal{A}$ be the attacker to the knowledge-soundness of $\mathcal{X}_{L1}$. Through $\mathcal{E}_a$, $\mathcal{E}_b$ and $\mathcal{A}$ we build an adversary $\mathcal{B} = (\mathcal{B}_a, \mathcal{B}_b)$ that can break the knowledge-soundness of $\mathcal{X}_a$ or $\mathcal{X}_b$. The proof proceeds through defining an auxiliary game **H**, where we show that **H** is computationally indistinguishable from the game for knowledge-soundness (called **G**) and the wining probability of the adversary $\mathcal{A}$ in **H** is negligible (≈ 0).

Game **H**: is the same as knowledge-soundness game, except that, we modify the winning condition by adding the condition:

$$\mathsf{Cond}^* : \quad \mathcal{R}_1(x_1, \rho, w_1) = 0 \quad \text{and} \quad \mathsf{Verify}_1(\mathsf{pp}_1, x_1, \rho, \pi_1) = 1$$

Then we prove two following claims:

- **Claim 1.** The game **H** is computationally indistinguishable from the game for knowledge-soundness (i.e., **G**), if $\mathcal{X}_a$ is knowledge-sound.
- **Claim 2.** The wining probability of $\mathcal{A}$ in the game **H** is negligible, if $\mathcal{X}_b$ is knowledge-sound.

We give the proofs in Supplement G, Lemmas 2 and 3.

The second layer As stated in Section 5.4, the missing part of our compiler is that it can only make the two argument systems communicate by their public inputs. The second layer of compilation solves this problem by allowing moving parts of the public inputs (called v) into the witness (see Fig. 11). Indeed, the aim is to delegate parts of the computation (involved with v) to the prover, without breaching the soundness.

Therefore, the second layer takes as input $\mathcal{X}_{L1}$, a public-coin single-round argument of knowledge for the relation $\mathcal{R}_{L2}$. Let F_a be the computation-step in the verification algorithm (Definition 12) of $\mathcal{X}_{L1}$ (inherited from $\mathcal{X}_a$)[8]. Let ξ, ζ be the partitioning of the public input and witness space (res.) and $\mathcal{X}_{\mathcal{T}}$ be an AoK for the relation $\mathcal{R}_{\mathcal{T}}(\mathsf{vk}, \xi) = \{(\gamma_v, v) \ : \ \gamma_v \leftarrow \Phi_L(v)\}$ where $(v; x_{L2}) = \xi(x)$ (and ϕ_L is the split left of F_a, as defined in Section 5.3). The second layer of compilation builds a SNARK for the relation,

$$\mathcal{R}_{L2} = \left\{ (x_{L2}; w, v) : \begin{matrix} x = \xi^{-1}(v, x_{L2}) \\ \mathcal{R}_{L1}(x, w) = 1 \end{matrix} \right\} = \left\{ (x_{L2}; w, v) : \mathcal{R}_{L1}(v, x_{L2}; w) = 1 \right\} \tag{3}$$

where the second equality holds thanks to the fact that ξ is a partitioning. The proof consists of tree main parts; the proof of the first layer, the value γ_v and a proof generated by $\mathcal{X}_{\mathcal{T}}$ as the AoK for the relation $\mathcal{R}_{\mathcal{T}}$. We also describe the relation built by the layer 2 in *Fig.* 11, where x and w are associated with layer 1, and we are moving v from x to w_{L2}.

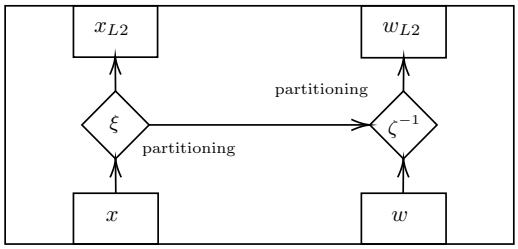

Fig. 11. Representation of the layer 2

Layer 2 Construction The inner-work of second layer of the compiler is given in Fig. 12. Here $(\mathsf{Prove}_{1,L1}, \mathsf{Prove}_{2,L1})$ stands for the prover algorithm of layer

[8] Note that if $\mathcal{X}_a$ has 2-step verification, then our,$\mathcal{X}_{L1}$ has this property as well.

1. Note that the verification algorithm of $L1$ has 2-step verification, here we use γ as the output of the computation-step and $\mathsf{Verify}'_{L1} = \mathsf{Verify}'_0$ as algorithm for the justification-step. Moreover, since F_a (the map in the computation-step) is compatible with the splitting ξ, we can denote (ϕ_L, ϕ_R) and g as the splitting and combiner for F_a (Definition 14). The last point is that, thought for clarity we use $\mathsf{pp}, \mathsf{pp}_{\mathcal{T}}$ in the setup (Fig. 12), indeed we have $\mathsf{pp} \subset \mathsf{pp}_{\mathcal{T}}$, this fact is particularly used in the security reduction.

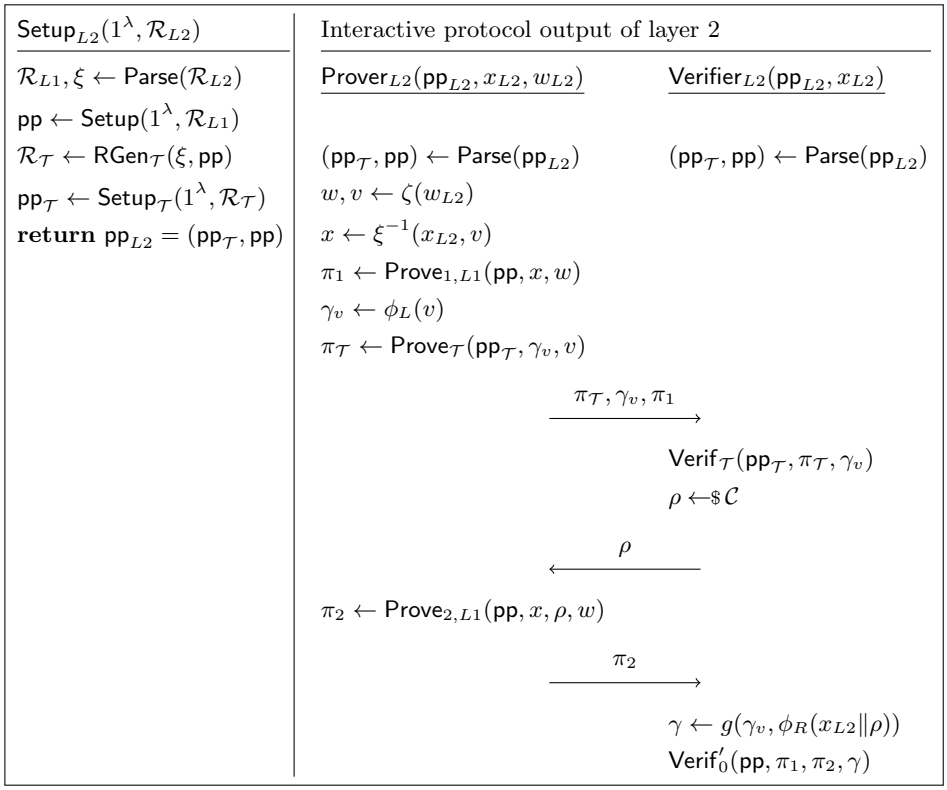

Fig. 12. Output protocol of the layer 2

Completeness: is straightforward from the completeness of $\mathcal{X}_{L1}$ and $\mathcal{X}_{\mathcal{T}}$.
Knowledge-Soundness.

Let $\mathcal{E}_T$ and $\mathcal{E}_{L1}$ be the extractors associated, respectively, with $\mathcal{X}_{\mathcal{T}}$ and $\mathcal{X}_{L1}$. If our argument system $\mathcal{X}_{L2}$ outputs a valid proof, to obtain a witness we should run the extractors $\mathcal{E}_{\mathcal{T}}$ and $\mathcal{E}_{L1}$ (to obtain v and w, respectively) and the probability that either of these extractors fails is negligible.

Theorem 2. *Let $\mathcal{X}_{L1}$ be a succinct argument of knowledge for a relation $\mathcal{R}_{L1}$ whose verifier has a 2-steps structure and such that it's computation-steps is*

compatible with all partitioning. (We do not require non-interactivity). If $\mathcal{X}_{L1}$ and $\mathcal{X}_{\mathcal{T}}$ both are knowledge-sound with knowledge-error ϵ_{L1} and $\epsilon_{\mathcal{T}}$ (res.), then the protocol $\mathcal{X}_{L2}$ is knowledge-sound with knowledge-error $\epsilon_{L2} = O(\epsilon_{L1} + \epsilon_{\mathcal{T}})$.

Our proof technique is similar to the one for Theorem 1. The difference is that to define the auxiliary game **H**, we modify the winning condition to $\mathcal{R}_{\mathcal{T}}(\gamma_v; v) = 0$. The proof is given in Supplement H. In Supplement F, we combine layers of compilation together and present the general form of the relation that our compiler deals with.

5.5 Instantiation of Fiat-Shamir

Here, we present a technical description of how we apply Fiat-Shamir to remove the interactive round of the protocol. We compile the protocol in the random oracle model. As it is a single-round protocol, the two models *so-far transcript* and *so-far digest* (see Supplement J yields the same protocol in the end. Therefore, it does not matter which one to pick. It would matter, however, if the present protocol is being used as part of a larger protocol. For that matter, we recommend designers of such protocols to use the interactive version of our protocol $\mathcal{X}_{L2}$ and then apply the Fiat-Shamir transform over this. Below, we discuss instatiation of fiat-shamir for our compiler as a solo-protocol (where our scheme is not part of a larger protocol).

The message to be sent to the random oracle must comprise

- The public parameters of $\mathcal{X}_{L2} = (\mathsf{pp}_T, \mathsf{pp}_{L1})$
- The prover message $\pi_{\mathcal{T}}, \gamma_v, \pi_1$
- The public inputs of the protocol x_{L2}

We then instantiate the random oracle with a hash function that can modeled as a random oracle. Here we do not require the hash function to be efficiently verifiable in an arithmetic circuit[9]. Thus, standard hash functions like Keccak, SHA2-256 or Blake2/3 can be used here.

As we explain in Supplement J, numerous hash functions, like the one we mention above, work by iteratively updating the state of a hash function before returning the result. This allows us to precompute the state of the hash function with the public parameters of $\mathcal{X}_{L2}$ at the end of the setup and *only hash* the public inputs of the protocol. This implementation detail is important because otherwise, one would have to hash the (possibly gigantic) public parameters of $\mathcal{X}_{L2}$ at every run of the protocol.

6 Instantiating the Compiler

Here, we give a specification for the choice of building blocks: $\mathcal{X}_a$ as the outer-layer SNARK, $\mathcal{X}_b$ as the public-coin single-round interactive argument of knowledge, and $\mathcal{X}_{\mathcal{T}}$ as AOK for the computation of γ_v. We instantiate $\mathcal{X}_a$ by Groth16

[9] Unlike the hash function in the single-round version of GKR that we describe in Supplement E.1 (i.e., GKR_s)

or Plonk and present the single-round GKR as the instantiation for $\mathcal{X}_b$. Finally, we give a concrete AOK (corresponding to $\mathcal{X}_\mathcal{T}$) compatible with the outer-layer SNARK, that is, $\mathcal{X}_a$. For both Groth16 [23] and Plonk [2], we present our AOK system $\mathcal{X}_\mathcal{T}$ separately. We emphasize that the AOK system $\mathcal{X}_\mathcal{T}$ is not general and is built with respect to the underlying $\mathcal{X}_a$.

6.1 Argument of knowledge $\mathcal{X}_\mathcal{T}$ for Groth16

Remember that in our layer 2, the prover sends a commitment γ_v to a part of public input (denoted as v) of underlying SNARK $\mathcal{X}_a$, and also an AOK that commitment is computed correctly. Here, we discuss how to build $\mathcal{X}_\mathcal{T}$ for the case that $\mathcal{X}_a$ is instantiated with Groth16. Note that under the algebraic group model [16], the Groth16 [23] protocol has witness extended emulation (thus, knowledge-soundness). It also has the requirement that $\mathcal{X}_a$ should satisfy; 2-step verification, where the public-input contribution F_a is a MSM and so is σ-compatible for any splitting σ. We provide a protocol $\mathcal{X}_\mathcal{T}$ as the AOK compatible with Groth16. We remind the reader that the aim of $\mathcal{X}_\mathcal{T}$ is to provide an AOK for the following relation;

$$\mathcal{R}_\mathcal{T}(\mathsf{vk}, \sigma) : \{(\gamma_v; v) \ : \ \gamma_v = \phi_L(v)\}$$

where vk is the Groth16 verification key, and ϕ_L is the left split of F_a (the computation step of Groth16 verification) according to the splitting ξ, and v comes from $\xi(x) = (x_{L2}, v)$ splitting of the Groth16's public input x.

The protocol $\mathcal{X}_\mathcal{T}$ is described in Fig. 13. Assume $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T; \mathbb{F})$ to be the description of a bilinear group, $n \in \mathbb{N}^*$ and $\mathsf{vk} \in {\mathbb{G}_1}^n$ the Groth16 verification key. The setup simply splits vk according to ξ and randomized vk_L. The proof is just a MSM of randomize vk_L and v.

$\underline{\mathsf{Setup}(\mathsf{pp}_{Groth16}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{F}, \xi)}$

$\sigma \leftarrow\!\$\, \mathbb{F}$

$g \leftarrow\!\$\, \mathbb{G}_2$

$(\mathsf{vk} \in \mathbb{G}_1, -) \leftarrow \mathsf{Parse}(\mathsf{pp}_{Groth16})$

$\mathsf{vk}_L, \mathsf{vk}_R \leftarrow \xi(\mathsf{vk})$

$\mathbf{L} \leftarrow \mathsf{vk}_L^\sigma$

return $\mathsf{srs} = (\mathsf{pp}_{Groth16}, g, g^{\frac{1}{\sigma}}, \mathbf{L})$

$\underline{\mathsf{Prove}(\mathbf{L}, v)}$

$\pi \leftarrow \mathsf{MSM}(\mathbf{L}, v)$

return π

$\underline{\mathsf{Verify}(\mathsf{vk}, \mathbf{L}, \pi, \gamma_v, g, g^{\frac{1}{\sigma}})}$

$e(\pi, g^{\frac{1}{\sigma}}) \stackrel{?}{=} e(\gamma_v, g)$

Fig. 13. Argument of Knowledge for Groth16

Note that the above AOK prevents the mix-and-match attack (see Section 5.1), since the toxic randomness σ is hidden. Slightly more in detail, if the adversary tries to mix γ_v with the rest of the verification equation of Groth16, to pass the verification check of Groth16, it can not pass the verification of $\mathcal{X}_\mathcal{T}$.

Now we prove the knowledge-soundness of $\mathcal{X}_\mathcal{T}$ in the algebraic group model.

Theorem 3. *Our Argument of Knowledge $\mathcal{X}_{\mathcal{T}}$ for Groth16 (Fig. 13) is knowledge-sound in the Algebraic Group Model [16] under the DLog assumption.*

Since we are in the algebraic model, the extractor simply receives the LC-representation of γ_v and π, which gives the witness. Based on the pairing check, these LC-representation should satisfy a special equality. We show that the only way that the adversary passes the equality check for a specific value σ is to break the DLog assumption. On the other hand, it cannot pass the check for arbitrary σ, since this imposes a special structure on the LC-representation which essentially forces the relation to hold (so the winning condition cannot be satisfied). The formal proof is given in Supplement I.

6.2 Argument of Knowledge $\mathcal{X}_{\mathcal{T}}$ for Plonk

The very popular PLONK [2] protocol is a zk-SNARK for arithmetic constraint systems. As described by its authors, it does not have the 2-step verification property. Thus, we cannot directly apply our compiler on the original PLONK protocol. Fig. 14 illustrates a very simplified view of how the Plonk verifier processes its public input. In the following, we borrow the notation of [2]. For an assignment vector w (meaning the concatenation of the public inputs with the witness in a single vector), the ℓ first entries $w_{i\leq\ell}$ denote the public inputs and $w_{i>\ell}$ the witness, $\mathsf{pp}_{\mathcal{P}\mathfrak{Plonk}}$ denotes the public parameters (including the preprocessed inputs) and $\pi_{\mathcal{P}\mathfrak{Plonk}}$ the proof. As in [2], L_i denotes the Lagrange polynomial for the root of unity ω^i on a larger subgroup of roots of unity Ω.

As explained in Fig. 14), at a high level, a randomness $\mathfrak{z}$, among other randomness, is obtained by hashing the verifier's input in (1). This randomness is used as evaluation point for the interpolation polynomial of the public inputs on Ω in (2). Informally, we could try to pick (1) and (2) as the computation-step and (3) as the justification-step (see Definition 12). This is unfortunately invalid: in order to compute $\mathfrak{z}$, we need to include the proof in the Fiat-Shamir hash (denoted by Hash), thus it is not trivial how to split the computation.

$\mathsf{Verify}(\mathsf{pp}_{\mathcal{P}\mathfrak{Plonk}}, w_{i<l}, \pi_{\mathcal{P}\mathfrak{Plonk}})$

// Implictly, test that the proof and the public
// input are valid fields and subgroup elements

1 : $\mathfrak{z}, \cdots$ other randomnesses $\leftarrow \mathsf{Hash}(\mathsf{pp}_{\mathcal{P}\mathfrak{Plonk}}, w_{i\leq\ell}, \pi_{\mathcal{P}\mathfrak{Plonk}})$

2 : $\mathsf{PI}(\mathfrak{z})) = \sum_{i\leq\ell} w_i L_i(\mathfrak{z})$

...

3 : $b \leftarrow \mathsf{OtherCheck}(\mathsf{pp}_{\mathcal{P}\mathfrak{Plonk}}, \mathsf{PI}(\mathfrak{z}), \pi_{\mathcal{P}\mathfrak{Plonk}})$

return b

Fig. 14. A very simplified description of the PLONK verifier

Our variant of Plonk To bypass the aforementioned problem, we consider a family of variants of Plonk instead of the original protocol itself. We use the notations of [2] to make it easier for the reader to determine the changes compared to the original scheme, although we give the equivalent notation of our compiler in Fig. 17. In Fig. 15 we represent our variant where the verifier does not directly evaluate $\mathsf{PI}(\mathfrak{z})$ as in (2) for [2]. Instead, he splits the public input into two parts : $w_{(1)}, w_{(2)} = \mathsf{Part}(w_{i\leq\ell})$ and computes the KZG commitment, $[\mathsf{PI}_2(x)]_1$ We denote by Q the subset of indices selected by Part for $w_{(1)}$. Part is to be considered as a parameter of the protocol.

Picking $\mathsf{Part} : w \to (w, \emptyset)$ yields the same protocol as Plonk. For $w_{(1)}, w_{(2)} = \mathsf{Part}(w_{i\leq\ell})$, it is similar to the PLONK, except that, instead of $w_{(2)}$, we use the commitment $[PI_2(x)]_1$ to generate the randomness $\mathfrak{z}$. As the verifier compute $PI_2(x)$ itself, from a security standpoint the variant described above is not different from PLONK, for any splitting Part. Implicitly, we will select $\mathsf{Part} = \xi$ (from Section 5.4) to make it compatible with our compiler. The last point is that though in [2] they use only the terms $[x^i]_1, [x]_2$ for $i = 0 \cdots n + 5$ (as pp), the security proof is based on $\mathsf{srs} = ([x^i]_1, [x^i]_2,\ i = 0 \cdots n + 5)$. Here we use the whole srs, since we use the elements of $[x^i]_2$ in our AOK system $\mathcal{X}_{\mathcal{T}}$.

Description of AOK $\mathcal{X}_{\mathcal{T}}$ for our variant of Plonk Our variant of Plonk is secure under the Algebraic Group Model [16] and has 2-step verification. The computation-step involves the evaluation of hashes and the polynomial $PI(X)$ over the point $X = \mathfrak{z}$. Through our compiler, we split this computation into two parts; for the hash computation, we use the trivial splitting given in Fig. 18 where the commitment is computed by the prover. For splitting $PI(X)$ over the point $X = \mathfrak{z}$, the verifier part is $PI_1(\mathfrak{z})$ and the prover part is $PI_2(\mathfrak{z})$. The prover should give a proof for the correct computation of its share.

Let ω_i's be the n-th roots of unity (such that for public inputs x we have $|x| \leq n$, where in our case $\xi(x) = (v, x_{L2})$, using our notation Fig. 17), we can derive three subsets $\omega_{L2}, \omega_v, \omega_{n\setminus\ell} \subset \Omega = \{\omega_i\}_i$ where the first two subsets correspond to x_{L2} and v, respectively.

One can consider $PI(X) = PI_1(X) + PI_2(X)$ such that the polynomial $PI_1(X)$ is handled by the verifier and over unity roots ω_{L2} interpolates to the public inputs of layer 2 (i.e. x_{L2}) while $PI_2(X)$ is handled by the prover and interpolates to v over the corresponding unity roots ω_v and to zero over $\omega_{L2} \cup \omega_{n\setminus\ell}$. Our compiler, outputs $\gamma_v = (C, y_{\mathfrak{z}})$ (where $C = \sum_i v_i [L_i(x)]_1$ is the KZG commitment to $PI_2(X) = \sum_i v_i L_i(X)$, and $y_{\mathfrak{z}} = PI_2(\mathfrak{z})$) and an AOK $\mathcal{X}_{\mathcal{T}}$, as given in Fig. 16, allowing the verifier to check that $PI_2(X) := V(X)$ evaluates to 0 on the entries $\omega_{L2} \cup \omega_{n\setminus\ell}$, and to $y_{\mathfrak{z}}$ at $X = \mathfrak{z}$. The former guarantees that C is computed using only the correct Lagrange basis, while the latter guarantees the correct computation of $PI_2(\mathfrak{z})$. The first part of the proof can be done with a batch opening of KZG polynomial commitment. Thus, the argument system $\mathcal{X}_{\mathcal{T}}$ is a batch opening of KZG at points $\omega_{L2} \cup \omega_{n\setminus\ell}$ and a single opening for $X = \mathfrak{z}$. In [7] the authors proved the knowledge-soundness of batch KZG opening in the algebraic group model and based on q-DLog assumption (Supplement A, Defini-

$\mathsf{Preprocess}(\mathcal{R}, [x^i]_1, [x^i]_2 : i = 0 \cdots n+5)$

for $i \in U$: $\quad [L_i(x)]_1 \leftarrow \sum_{k<|\Omega|} L_{i,k} \cdot x^k \cdot [1]_1$

$\mathsf{pp}_{\mathcal{P}\mathfrak{lonk}} \leftarrow \mathsf{Preprocess}_{\mathcal{P}\mathfrak{lonk}}(\mathcal{R}, \{[x^i]_1, [x^i]_2 : i = 0 \cdots n+5\})$

return $\mathsf{pp}' = (\mathsf{pp}_{\mathcal{P}\mathfrak{lonk}}, [x^j]_2 \{[L_i(x)]_1 : i \in U, j = 0 \cdots n+5\})$

$\mathsf{Prove}(\mathsf{pp}', w)$	$\mathsf{Verify}(\mathsf{pp}', w_{i<l}, \pi')$
// We add the following steps at the beginning	$[\mathsf{PI}_2(x)]_1 \leftarrow \sum_{i \in U} w_i \cdot [L_i(x)]_1$
$w_{(1)}, \_ \leftarrow \mathsf{Part}(w_{i<l})$	$\cdots$
$[\mathsf{PI}_2(x)]_1 \leftarrow \sum_{i \in U} w_i \cdot [L_i(x)]_1$	$\mathfrak{z} \leftarrow \mathsf{Hash}(\text{Transcript})$
$\cdots$	$\mathsf{PI}_1(\mathfrak{z}) \leftarrow \sum_{i \in Q} w_i L_i(\mathfrak{z})$
// Run Plonk's prover rounds 1, 2, 3	$\mathsf{PI}(\mathfrak{z}) = \mathsf{PI}_1(\mathfrak{z}) + \mathsf{PI}_2(\mathfrak{z})$
// For the Fiat-Shamir hashes use	$\cdots$
// $[\mathsf{PI}_2(x)]_1$ is place of $w_{(2)}$	// Resume with the normal Plonk verifier
$\pi_{1,2,3} \leftarrow \cdots$	
$\cdots$	
// Then, at round 4	
$\mathfrak{z} \leftarrow \mathsf{Hash}(\text{Transcript})$	
$\cdots$	
// Resume with the original Plonk prover	
$\cdots$	
$\pi_{4,5} \leftarrow \cdots$	
return $(\pi_{1,2,3}, \pi_{4,5})$	

Fig. 15. A variant of the Plonk protocol

tion 15). Here, our srs is longer but all the extra terms are linear combinations

$\mathsf{Setup}(\mathsf{pp}', \mathbb{G}_1, \mathbb{G}_2, \mathbb{F}, \xi)$	$\mathsf{Prove}(\mathbf{L}, v, \mathfrak{z})$	$\mathsf{Verify}(\pi, \pi', C, y_{\mathfrak{z}}, g_1^x, \mathbf{Z})$
$[L_i(x)]_1, [x^j]_2 \leftarrow \mathsf{pp}'$	$v(X) = \sum v_i \cdot L_i(X)$	$e(C, [1]_1) \stackrel{?}{=} e(\pi, [x - \mathfrak{z}]_2)$
$Z(X) \leftarrow \prod_{i \in T} (X - \omega^i)$	$y_{\mathfrak{z}} \leftarrow v(\mathfrak{z})$	$e(C, [1]_1] \stackrel{?}{=} e(\pi', \mathbf{Z})$
$\mathbf{Z} \leftarrow Z(x) \cdot [1]_2$	$\pi \leftarrow \mathsf{KZG.Prove}(v(X), y_{\mathfrak{z}}, \mathfrak{z})$	
return	$\pi' \leftarrow \mathsf{KZG.Prove}(v(X), \vec{0}, \{\omega^i\}_{i \in T})$	
$\mathsf{srs} = (\mathsf{pp}', \mathbf{L} = ([L_i(x)]_1), \mathbf{Z})$	**return** (π, π')	

Fig. 16. Argument of Knowledge for PLONK. Here pp' is the public parameter of our variant in Fig. 15, and T is the set of indices associated with the unity roots in $\omega_{L2} \cup \omega_{n \setminus \ell}$

of elements in the original srs in [7], thus the security still holds in the algebraic group model.

7 Implementation and Performance Analysis

We provide a parallel open-source implementation[10] of the protocol in Golang. The implementation is optimized and benchmarked for massive parallelism : the protocol has been benchmarked over AWS hpc6a instances (96 physical cores and 384 Gb of memory). The implementation uses the libraries **gnark**[11] and **gnark-crypto**[12] for the finite field arithmetic and the Groth16 implementation. Various kind of optimizations have been carried out : lowering the overheads of parallelization, pooling the memory to reduce the overheads of allocations, reducing the number field of arithmetic operations.

In Fig. 19, we give the results of benchmarks measuring the speed at which our implementation can prove MiMC permutations. The benchmarks are performed using the curve BN254 [3] and the results are presented in *Fig.* 19. As a point of comparison, we have benchmarked the prover time of a circuit performing multiple MiMC permutations using gnark's implementation of Groth16 (without using GKR). For a number 2^{18} of MiMC permutations, it runs in 38.3 sec. This corresponds to proving 6835 hashes per second. Our techniques also brings a small improvement in memory usage between the two approaches, but it is much smaller. That is because the GKR prover still need writing the set of all the intermediate values at the same time.

[10] https://github.com/ConsenSys/gkr-mimc
[11] https://github.com/ConsenSys/gnark
[12] https://github.com/ConsenSys/gnark-crypto

Scheme	public input	witness	verifier part	Prover part	Prover contribution
PLONK	$\{w_i\}_{i\in[\ell]}$	$\{w_i\}_{i=\ell+1}^{3n}$	$\{w_i\}_{i\in[Q]}$	$\{w_i\}_{i\in[\ell\setminus Q]}$	C, y_ζ
Ours	$x = \xi(v, x_{L2})$	w	x_{L2}	v	γ_v

Fig. 17. Notation-Equivalence for PLONK and our scheme

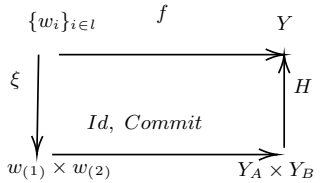

Fig. 18. Splitting for hash computation; Id is the identical function and Commit is the KZG commitment.

Number of Hashes	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}
Initial Randomness	120 ms	196 ms	412 ms	756 ms	1293 ms	2504 ms
Gkr Prover	3.4 s	5.4 s	13.1 s	19.7 s	29.1 s	51.8 s
Groth16 Prover	4.0 s	6.4 s	7.1 s	12.5 s	21.3 s	24.5 s
Total (Gkr)	7.6 s	12.0 s	20.7 s	33.0 s	51.7 s	78.9 s
Hash per second	68400	86500	101000	127000	162000	212000

Fig. 19. Runtime efficiency benchmarks for GKR

8 Acknowledgements

We thank Dan Boneh for pointing out a flaw in an earlier version of the protocol and for insightful discussions. We are also grateful to Gautam Botrel, Youssef El Housni, Arya Tabaie, Gus Gutoski, Thomas Piellard, Vanessa Bridge and Nicolas Liochon for their feedback and useful discussions on the protocol itself, the paper and its implementation.

References

[1] Martin R. Albrecht et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity". In: *ASIACRYPT*. Vol. 10031. LNCS. 2016, pp. 191–219.

[2] Zachary Ariel Gabizon, J. Williamson, and Oana Ciobotaru. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". In: *IACR Cryptol. ePrint Arch.* (2019), p. 953.

[3] Paulo Barreto and Michael Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order". In: *LNCS*. Vol. 3897. Aug. 2005, pp. 319–331.

[4] Eli Ben-sasson, Alessandro Chiesa, and Nicholas Spooner. "Interactive Oracle Proofs". In: *Theory of Cryptography TCC 2016-B*. Vol. 9986. LNCS. 2016, pp. 31–60.

[5] Eli Ben-sasson et al. "Scalable Zero Knowledge Via Cycles of Elliptic Curves". In: *Algorithmica* 79 (Oct. 2016), pp. 1–59.

[6] John Black, Phillip Rogaway, and Thomas Shrimpton. "Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV". In: *Advances in Cryptology - CRYPTO 2002*. Vol. 2442. LNCS. Springer, 2002, pp. 320–335.

[7] Dan Boneh et al. "Efficient polynomial commitment schemes for multiple points and polynomials". In: *IACR Cryptol. ePrint Arch.* (2020), p. 81.

[8] Sean Bowe, Jack Grigg, and Daira Hopwood. *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Report 2019/1021. 2019.

[9] Benedikt Bünz et al. "Proofs for Inner Pairing Products and Applications". In: *Advances in Cryptology - ASIACRYPT*. Vol. 13092. LNCS. Springer, 2021, pp. 65–97.

[10] Vitalik Buterin. *On-chain scaling at potentially 500 transactions per seconds*. ethresearch/3477. 2019.

[11] Matteo Campanelli, Dario Fiore, and Anaïs Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". In: *ACM SIGSAC*. Nov. 2019, pp. 2075–2092.

[12] Ran Canetti et al. "Fiat-Shamir From Simpler Assumptions". In: *IACR Cryptol. ePrint Arch.* (2018), p. 1004.

[13] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. "Fractal: Post-quantum and Transparent Recursive Proofs from Holography". In: *LNSC*. Vol. 12105. May 2020, pp. 769–793.

[14] Alessandro Chiesa et al. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: *Advances in Cryptology EUROCRYPT*. Vol. 12105. LNCS. Springer, 2020, pp. 738–768.

[15] Arka Choudhuri et al. "Finding a Nash equilibrium is no easier than breaking Fiat-Shamir". In: *ACM, STOC*. 2019, pp. 1103–1114.

[16] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. "The Algebraic Group Model and its Applications". In: *Advances in Cryptology - CRYPTO*. Vol. 10992. LNCS. Springer, 2018, pp. 33–62.

[17] Ariel Gabizon and Zachary J. Williamson. "Plookup: A simplified polynomial protocol for lookup tables". In: *IACR Cryptol. ePrint Arch.* (2020).

[18] Ariel Gabizon and Zachary J. Williamson. *Proposal: The Turbo-PLONK program syntax for specifying SNARK programs*. zkproof.org. `https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf`. 2020.

[19] Oded Goldreich. "On Doubly-Efficient Interactive Proof Systems". In: *Foundations and Trends® in Theoretical Computer Science* 13 (Jan. 2018), pp. 157–246. DOI: `10.1561/0400000084`.

[20] Shafi Goldwasser, Yael Kalai, and Guy Rothblum. "Delegating Computation: Interactive Proofs for Muggles". In: *ACM STOC*. ACM, 2008, pp. 113–122.

[21] Lorenzo Grassi et al. "On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy". In: *Advances in Cryptology - EUROCRYPT*. Vol. 12106. LNCS. Springer, 2020, pp. 674–704.

[22] Lorenzo Grassi et al. "Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems". In: *IACR Cryptol. ePrint Arch.* (2019), p. 458.

[23] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology - EUROCRYPT*. Vol. 9666. LNCS. Springer, 2016, pp. 305–326.

[24] Daira Hopwood et al. *Zcash protocol specification: Version 2022.04.26 Technical report, Zerocoin Electric Coin Company*. `https://github.com/zcash/zips/blob/main/protocol/protocol.pdf`. 2022.

[25] Aniket Kate, Gregory Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *Advances in Cryptology - ASIACRYPT*. Vol. 6477. LNCS. Springer, 2010, pp. 177–194.

[26] Yehuda Lindell. "Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation". In: *Advances in Cryptology - CRYPTO*. Vol. 2139. LNCS. Springer, 2001, pp. 171–189.
[27] Carsten Lund et al. "Algebraic Methods for Interactive Proof Systems". In: *FOCS*. IEEE Computer Society, 1990, pp. 2–10.
[28] Mary Maller et al. "Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings". In: *ACM SIGSAC -CCS*. ACM, 2019, pp. 2111–2128.
[29] Bart Preneel, René Govaerts, and Joos Vandewalle. "Hash Functions Based on Block Ciphers: A Synthetic Approach". In: *Advances in Cryptology - CRYPTO '93*. Vol. 773. LNCS. Springer, 1993, pp. 368–378.
[30] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. CRYPTO. 2020.
[31] Polygon Zero Team. *PLONKY2 : Fast Recursive Argument with Plonk and FRI*. https://github.com/mir-protocol/plonky2/blob/main/plonky2.pdf. Draft : version 2022. 2022.
[32] Riad Wahby et al. "Doubly-Efficient zkSNARKs Without Trusted Setup". In: *SP*. IEEE Computer Society, 2018, pp. 926–943.
[33] Riad Wahby et al. "Full Accounting for Verifiable Outsourcing". In: *ACM SIGSAC -CCS*. ACM, 2017, pp. 2071–2086.
[34] Tiacheng Xie et al. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: *Advances in Cryptology - CRYPTO*. Vol. 11694. LNCS. Springer, 2019, pp. 733–764.
[35] Jiaheng Zhang et al. "Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time". In: *ACM SIGSAC -CCS*. ACM, 2021, pp. 159–177.
[36] Jiaheng Zhang et al. *Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof*. 2020 IEEE Symposium on Security and Privacy (SP). 2019.
[37] William Zhang and Yu Xia. *Hydra: Succinct Fully Pipelineable Interactive Arguments of Knowledge*. Cryptology ePrint Archive, Report 2021/641. 2021.

Supplementary materials

A Background

The knowledge-soundness of Groth16 and Plonk is proved based on q-DLog assumption defined in the following.

Definition 15 (q-Discrete Log Problem [7, 2]). *Fix an integer q. The q-DLog assumption for $(\mathbb{G}_1, \mathbb{G}_2)$ states that, given*

$$[1]_1, [x]_1, \ldots, [x^q]_1; [1]_2, [x]_2, \ldots, [x^q]_2$$

for uniformly chosen $x \in \mathbb{F}_p$, the probability that a p.p.t adversary $\mathcal{A}$ outputs x is negligible.

Polynomial Commitment. Here we define the correctness and the security of a polynomial commitment scheme.

Definition 16 (Correctness). *We say that a polynomial commitment scheme has (perfect) completeness if for all* $P(X), t, \lambda, x$,

$$\Pr\left[1 \leftarrow \mathsf{Verify}(\mathsf{pp}, C, y, \pi) \ : \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, t) \\ C \leftarrow \mathsf{Commit}(\mathsf{pp}, P(X)) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, P(X), x) \\ y = P(x) \end{array}\right] = 1$$

Definition 17 (Secure polynomial commitment [7]). *A polynomial commitment (*Setup, Commit, Prove, Verify*) is secure if it is knowledge-sound w.r.t the relation,*

$$\mathcal{R} = \{(x, y; P(X)) \ : \ P(x) = y\}$$

The original paper [25] introduces the notion of polynomial commitment. In [7, 2], the authors gave a general security notion supporting batching of the proofs and well-detailed for interactive public-coin polynomial commitments. We expand on this in Supplement D.

Both [2] and [9] provide an analysis of an extended version of the KZG protocol under the algebraic group model [16] in which the author demonstrate knowledge soundness [2] and bounded-polynomial extractability [28] of the protocol. Both these results are achieved using the q-$\mathcal{DL}$ assumption.

B Description of the GKR protocol

The GKR protocol can be described as follows : the prover starts with a description of an arithmetic circuit $\mathcal{C}$ and a batch assignment $\mathcal{B}$ to $\mathcal{C}$. The verifier starts with only the input and output polynomials, (P_I) and (P_O). Our description of the GKR protocol is organized in two steps. First, we present a set of "mini-protocol" and then we give the full description in Fig. 20 using the mini-protocols.

Mini-protocol 1 : multi-claim reduction The prover wants to convince the verifier that, given a polynomial P, it holds that $P(x_i) = y_i$ for k couples $(x_i, y_i) \in \mathbb{K}^2$. The verifier initiates the protocol by sending a random challenge $\tau \leftarrow\!\!\$\, \mathbb{K}^k$. The prover and the verifier then engage in a sumcheck for the sum relation

$$\sum_{i<k} \tau_i y_i = \sum_{h \in \{0,1\}^n} (\sum_{i<k} \tau_i \mathsf{eq}(x_i, h)) P(h)$$

In the last step of the sumcheck, the verifier is interested in verifying a claim in the form, for some α and r that were established during the sumcheck.

$$\alpha = \sum_{i<k} \tau_i \mathsf{eq}(x_i, r)) P(r)$$

Instead of letting the verifier directly evaluate $P(r)$, the prover directly sends a claim of $p = P(r)$. The verifier can then verify that the claimed value of p is consistent with the above claim. The protocol then outputs the claim $P(r) = p$. At the end of the protocol, the verifier does not learn that $P(x_i) = y_i$. But we have that if any of these claim was wrong, then with overwhelming probability $P(r) \neq p$, hence the name "claim reduction".

Mini-protocol 2 : single-claim reduction The prover is given an assignment $\mathcal{B}$ to $\mathcal{C}$. The verifier is input a claim that for some v, not an input gate, and $P_v = \mathcal{B}(v)$, we have $P_v(x) = y$. The aim of the mini-protocol 2 is to reduce the verifier's claim into a collection of evaluation claims $P_{u,j}(r) = \alpha_u$ for all $u \in I(v)$. As for the mini-protocol 1, this is achieved using a sumcheck protocol, but over the relation given in Remark 3.

$$y = \sum_{h \in \{0,1\}^n} \mathsf{eq}(x, h) R_v(P_{u,0}(h), \ldots)$$

where R_v is the low-degree polynomial associated to v in the circuit. The rest goes as in the mini-protocol 1. The protocol outputs claims of the form $p_u = P_u(r)$ and we have that if $y \neq P_v(x)$, then with overwhelming probability, one of the $p_u \neq P_u(r)$ is wrong.

The full GKR protocol The verifier samples a random $\rho \leftarrow\!\!\$\, \mathbb{K}^n$. He then computes the evaluations of all $V_O(\rho)$ and sends ρ to the prover. The prover and the verifier then engage in an iterative process, going through each vertex of the arithmetic circuit in reverse order. Each step of the process aims at reducing the claims, made on the $\mathcal{B}(v)$, to the claims on children vertices of v.

C Custom gates for Poseidon

Poseidon [22] applies the Hades [21] strategy to build a SNARK friendly hash function. It uses the same S-box $x \to x^\alpha$ as in MiMC, but is optimized to permute messages from multiple field elements (here $n + 1$) at once. At a high level, it works by alternating heterogeneous nonlinear layers in which either only one of the field elements is passed through the S-Box (partial rounds) : $(x_0, x_1, \cdots, x_n) \to ({x_0}^\alpha, x_1, \cdots, x_n)$ or in which all field elements are passed through the S-Box elementwise (full rounds) : $(x_0, x_1, \cdots, x_n) \to ({x_0}^\alpha, {x_1}^\alpha, \cdots, {x_n}^\alpha)$. The non-linear layers are interleaved with linear layers instantiated by an MDS matrix.

Although our implementation does not include these custom gates for the Poseidon permutation, we describe below a list of custom gate to allow obtaining a GKR variant for the Poseidon permutation.

The S-box for partial-rounds : at the beginning of a partial-S-box layer, the prover and the verifier both have preemptively *agreed* on a set of evaluation claims for the polynomial $\mathcal{B}(v_0), \cdots, \mathcal{B}(v_n)$ representing each of the outputs of

GKR protocol highlines

```
  // initialize the claim register, it maps each vertex to a list of claim
claims := {}
for each output gate v_O :
  append claims[v_O] with V_O(ρ)
  // in reverse order
for v ∈ V(G) :
  if |claims(v)| > 1 :
    claim' ← miniProtocol1(v, claims(v))
  else :
    claim' ← claims(v)
  // input gate
  if I(v) = ∅ :
    verifier checks directly claim'
    continue
  newclaims ← miniProtocol2(v, claim)
  for {u, claim_u} ∈ newclaims :
    append claim[u] = claim_u
```

Fig. 20. GKR protocol

the partial-round functions. As only the first entry is modified by the round function, only it needs to be reduced to a claim on the (unique) polynomial $\mathcal{B}(u_0)$ for $I(v_0) = \{u_0\}$. For this, we use a GKR round with polynomial $R(x_0) = x^\alpha$. We can then retain all other claims on $\mathcal{B}(v_{i>0})$ for the next round.

The S box for full rounds : Here, we need to do a sumcheck for every claim. At a high level, we suggest reusing the same sumcheck as for the partial-rounds for all entries and batch them using a random linear combination. Equivalently, the verifier samples $(r_0, \cdots, r_n) \leftarrow\$ \mathbb{F}^{n+1}$ public coins and engages with the prover in the following sumcheck. We note $I(v_i) = \{u_i\}$ since they all can have only one single input vertex.

$$\sum_{i \leq n} r_i \mathcal{B}(v_i)(x) = \sum_{h \in \{0,1\}^N} \mathsf{eq}(x,h) \left[\sum_{i \leq n} r_i \mathcal{B}(u_i)(h)^\alpha \right]$$

The MDS layers : here we can directly convert the claim (without the claims) since all operations are linear and since MDS matrices are invertible and the one we use are typically small : typically at most 16x16 in real world application. Namely, if we note M to be the MDS matrix, then we can obtain claims for the

input vertices by multiplying the vector of current claim (properly ordered) by M^{-1}.

D Batched KZG commitment

In this section, we recall the batched version of the KZG polynomial commitment of [2, 7]. Their work extend the one of the original authors [25] and allows for multipoint opening for multiple polynomial.

Formally, given a finite field $\mathbb{F}$ of prime order and an integer $n \in \mathbb{N}$, c we consider a set of polynomial $\{P_0(X), \cdots, P_{n-1}(X)\}$, a set of evaluation points $T = \{x_0, x_1, \cdots x_t\}$, a set of evaluation point $S_i \subset T$ and $r_i : S_i \to \mathbb{F}$, the minimal degree polynomial mapping each evaluation point S_i to a claimed value for the polynomial P_i. For any subset $S \subset \mathbb{F}$, Z_S denotes the polynomial $\prod_{s \in S}(X - s)$.

As in Section 2.3, the protocol satisfies the completeness and computational knowledge-soundness property in the algebraic group model under the $Q - Dlog$ assumption. We give a description of the (interactive form) of the protocol in Fig. 21. The protocol can be transformed in an interactive protocol through the Fiat-Shamir transforms, as it is public-coin and has a constant number of rounds.

E One-round GKR

In this section, we explicitly specify a single-round version of the GKR protocol. We then carefully discuss its security in the random oracle model step-by-step, motivate our technical choices and finally explain how to instantiate it using the Fiat-Shamir transform. We juggle with several variants of the GKR protocol. The reader can assume that GKR stands for the original interactive GKR protocol and GKR_s for the single-round version of the GKR protocol. Other intermediate versions will be introduced further in the section when appropriate (see also Fig. 22). We emphasize that GKR_s is a precise instantiation of our one-round GKR that uses a deterministic hash function (starting from the second round). We explain how we choose the hash function in Supplement E.3.

E.1 The single-round version of GKR in the random oracle

We first describe $\mathsf{GKR}_{s,RO}$ which is a one-round GKR in the random oracle model. We specify how the protocol's challenges are obtained.

- The initial challenge ρ_1 is a genuine public-coin sampled by the verifier
- The second challenge ρ_2 is obtained by querying the random oracle with inputs : the last prover message together with ρ_1. In particular, the random oracle is not sent the GKR statement x at this round.

  $\vdots$

$\underline{\mathsf{Setup}(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, Q)}$

$x \leftarrow\$\, \mathbb{F}$

$\mathsf{pp} \leftarrow ([x^k]_1, [x^k]_2)_{i \leq Q}$

return pp

$\underline{\mathsf{Commit}(\mathsf{pp}, P(X) = \sum_{i \leq Q} p_i X^i)}$

$[P(x)]_1 \leftarrow \sum_{i \leq Q} p_k [x^k]_1$

return $[P(x)]_1$

Batch Evaluation Opening and Verification

$\mathsf{Prove}(\mathsf{pp}, \{[P_i(x)]_1, S_i, r_i, P_i : \ i < n\})$		$\mathsf{Verify}(\mathsf{pp}, \{[P_i(x)]_1, S_i, r_i : \ i < n\})$
	$\xleftarrow{\gamma \leftarrow\$\, \mathbb{F}}$	
$P(X) \leftarrow \sum_{i<n} \gamma^i Z_{T \setminus S_i}(X)(P_i(X) - r_i(X))$		
$h(X) \leftarrow \frac{P(X)}{Z_T(X)}$		
	$\xrightarrow{[h(x)]_1}$	
		$\forall i < n : \ Z_i \leftarrow [Z_{T \setminus S_i}(x)]_2$
		$F \leftarrow \prod_{i<n} e(\gamma^i [P_i(x)]_1 - [r_i(x)], [Z_i]_2)$
		$F \stackrel{?}{=} e([h(x)]_1, [Z_T(x)]_2)$

Fig. 21. Batching of KZG

– The i-th challenge ρ_i is obtained by querying the random oracle with the last prover message together with the previous challenge ρ_{i-1}

The reader can see that the present protocol $\mathsf{GKR}_{s,RO}$ is somewhat different from a standard compilation in the random oracle model. Two points are important here. First, we use a *so-far digest* (e.g, sending the previous challenge and the last message) construction while the compiling in the Random Oracle Model requires to hash *all the transcript so far*. Secondly, the protocol $GKR_{s,RO}$ retains an initial genuine interactive round while a common (FS-) transform of a protocol in the random oracle model does not "let even a single round" of protocol to remain. While the security of this construction can seem somewhat intuitive, its security must studied with great care.

E.2 Analysis in the random oracle model

Before moving forward with the concrete security, we first introduce Lemma 1 which will be used to justify the second point. In the following theorem we use the intermediate GKR-versions introduced in Fig. 22.

Lemma 1. *Our single-round GKR (i.e.* $\mathsf{GKR}_{s,RO}$*) is sound in the random oracle model, if non-interactive* GKR *(i,e,* GKR_b*) is sound in the so-far digest random oracle model.*

Proof. Let $\mathcal{A}$ and $\mathcal{B}$ be respectively the attackers to the soundness of non-interactive GKR and single-round GKR. The adversary $\mathcal{A}$ simulates the soundness game for $\mathcal{B}$.

- when $\mathcal{B}$ sends x, the adversary $\mathcal{A}$ sends a RO-query to $H_{\mathcal{FS}}$ for the input x. It set the response as the challenge (generated by the verifier) for the first round and sends it to $\mathcal{B}$
- If $\mathcal{B}$ queries the random oracle $H_{\mathcal{FS}}$ for the same value x, the adversary $\mathcal{A}$ responds with a different random value.
- When $\mathcal{B}$ outputs a proof π, the adversary $\mathcal{A}$ outputs x and π.

Note that by this simulation, both protocols are technically the same but as $\mathcal{B}$ does not understand what it received at the first round was indeed a response to a RO-query over x (asked by the verifier, namely $\mathcal{A}$ here). From its perspective, the adversary $\mathcal{B}$ is inside an interactive version.

The security analysis consists in a sequence of transformation that starts from the original GKR protocol finish $\mathsf{GKR}_{s,RO}$. We summarize it in the table Fig. 22.

Protocol	Details	Argumentation
GKR	The original protocol, fully interactive	-
GKR_a	In the random oracle model, so-far transcript model	Standard compilation in ROM
GKR_b	In the random oracle model, so-far digest model	Theorem 4
$\mathsf{GKR}_{s,RO}$	Supplement E.1	Lemma 1
GKR_s	instantiated one-round GKR Supplement E.3	Fiat-Shamir hash function

Fig. 22. Overview of the security reduction

E.3 Instantiation of the random oracle

We apply the Fiat-Shamir heuristic to instantiate the random oracle. We do so, by selecting a hash function such that:

- It behaves like a random oracle
- It can be efficiently verified in an arithmetic circuit
- When the construction is applied for proving hashes, it is preferable to either use another hash function (different from the one that is being proved) or at least to change the parameters of the hash functions.

In our implementation, we use MiMC with different constants. We stress that at this step the transform only applies for the calls to the random oracle

and not *the genuine random-coin* of the first round. Though later at the third layer of our compiler (Section 5.4), we again use the fiat-shamir heuristic to instantiate the first round interaction with a hash function (again MiMc with different parameters). Indeed, what we are using in the compiler in (Section 5.4) is GKR_s.

F Plugging the layers together

Here we just put together both layers to see the general form of the relation. Let $\mathcal{R}_0$ and $\mathcal{R}_1$ be the relation associated to (respectively) $\mathcal{X}_a$ and $\mathcal{X}_b$ (defined in Section 5.4). Our compiler can output a proof system for relations of the form

$$\mathcal{R}'(\mathcal{R}_0, \mathcal{R}_1, \sigma_x, \sigma_w, \sigma_1, \kappa) = \left\{ (x' \in \mathbb{F}^{l'}; w' \in \mathbb{F}^{n'}) : \begin{array}{ll} w_{L1}, v = \zeta(w') & \\ x', v = \xi(x_{L1}) & \mathcal{R}_0(x_0; w_0) = 1 \\ x_0, x_1 = \sigma_t(x_{L1}) & \mathcal{R}_1(x_1; w_1) = 1 \\ w_0, w_1 = \sigma_u(w_{L1}) & \end{array} \right\}$$

We reuse the notations for the splittings introduced in Section 5.4. To apply the compiler, one needs to select the appropriate $\sigma_t, \sigma_u, \xi, \zeta$. To illustrate how to do so, we give a concrete example of how an assignment for the target relation $\mathcal{R}'$ would be structured. Then we clarify how this relates to the informal description of the construction in Section 5.1.

We first conveniently split the complete assignment (x', w') fine-grained way as,

- θ_C : the public inputs exclusive to $\mathcal{R}_0$
- θ_G : the public inputs that are shared by $\mathcal{R}_0$ and $\mathcal{R}_1$
- ψ_G : the public inputs of $\mathcal{R}_1$ shared with the witness of $\mathcal{R}_0$
- ψ_C : the part of the witness exclusive to $\mathcal{R}_0$
- w_1 : the (entire) witness of $\mathcal{R}_1$

To obtain a proof system for $\mathcal{R}'$, one can then successively apply the layer 1 and 2 compilers Section 5.4 and set the following: $\sigma_t(x_{L1}) = (\theta_C \| \theta_G, \theta_G \| \psi_G)$, $\sigma_u(w_{L1}) = (\psi_C, w_1)$, $\xi(\theta_C \| \theta_G \| \psi_G) = (\theta_C \| \theta_G, \psi_G)$, $\zeta(\psi_C \| \psi_G \| w_1) = (\psi_G, \psi_C \| w_1)$.

G Proof of Theorem 1

Lemma 2. *Claim 1 is true.*

Proof. Consider the distinguisher $\mathcal{D}$ which aims to distinguish the games **H** from **G**. We build an adversary $\mathcal{B}_a$ that simulates the games for $\mathcal{D}$ and use the responses from $\mathcal{D}$ to attack the knowledge-soundness of $\mathcal{X}_a$. The simulation is as follows:

- $\mathcal{B}_a$ receives the parameters pp' for the argument system $\mathcal{X}_a$, runs the setup for $\mathcal{X}_b$ to get pp_1 and then sends $\mathsf{pp} = (\mathsf{pp}', \mathsf{pp}_1)$ to $\mathcal{D}$.

- the adversary $\mathcal{D}$ responds by $x = (x_0, x_1), \pi_{1,1}$, and $\mathcal{B}$ sends the randomness ρ.
- when the adversary $\mathcal{D}$ responds with π', the adversary $\mathcal{B}_a$ outputs $(x, \pi) = (x_0, x_1, \rho; \pi')$.

We define the event Bad^* where the Cond^* is not satisfied. This means if Bad^* happens with probability ϵ, then $\mathcal{D}$ can distinguish two games with the same probability. But the probability that Bad^* happens is negligible by knowledge-soundness of $\mathcal{X}_a$ (i.e., $\epsilon \leq \epsilon_a$ where ϵ_a is the knowledge error for $\mathcal{X}_a$). The reason that the outcome Bad^* breaks the knowledge-soundness of $\mathcal{X}_a$ is due to the fact that

- if $\mathcal{R}_1(x_1, \rho; w_1) = 1$ since the output of $\mathcal{D}$ satisfies $\mathcal{R}_{L1}(x, w) = 0$, we conclude that one of the equalities in $\mathcal{R}'$ associated with $\mathcal{X}_a$ is not satisfied and therefore $\mathcal{B}_a$ has an admissible tuple (x_0, x_1, ρ, π') to break the knowledge-soundness of $\mathcal{X}_a$.
- if $\mathsf{Verify}_1(\mathsf{pp}_1, x_1, \rho, \pi_1) = 0$, clearly, this means $\mathcal{R}'$ is not satisfied, which again gives an admissible output to break the knowledge-soundness of $\mathcal{X}_a$.

Lemma 3. *Claim 2 is true.*

Proof. To prove that the wining probability of $\mathcal{A}$ is the game $\mathbf{H}$ is negligible, we build an attacker $\mathcal{B}_b$ that simulates the game $\mathbf{H}$ for $\mathcal{A}$ and uses the responses from $\mathcal{A}$ to attack the knowledge-soundness of $\mathcal{X}_b$. The simulation is as follows.

- $\mathcal{B}_b$ receives the parameters pp_1 and runs the setup for $\mathcal{X}_a$ to get pp'. Then it sends $\mathsf{pp} = (\mathsf{pp}', \mathsf{pp}_1)$ to $\mathcal{A}$.
- $\mathcal{A}$ responds by $x = (x_0, x_1), \pi_{1,1}$, and $\mathcal{B}_b$ sends $x_1, \pi_{1,1}$ to its challenger and relays the randomness ρ.
- When $\mathcal{A}$ responds with π', the adversary $\mathcal{B}_b$ runs the extraction $\mathcal{E}_a$ to extract the witness $(w_0, \pi_{1,1}, \pi_{1,2})$. Finally, it outputs $\pi_{1,2}$.

By the definition of the game $\mathbf{H}$ and Cond^*, the output $(x_1, \rho; \pi_1)$ is an admissible output for the knowledge-soundness of $\mathcal{X}_b$. This concludes the knowledge-soundness of $\mathcal{X}_{L1}$ w.r.t the extractor $(\mathcal{E}_a, \mathcal{E}_b)$ and with knowledge error $\epsilon_{L1} \leq \epsilon_a + \epsilon_b$.

H Proof of Theorem 2

Proof. Let $\mathcal{A}$ be the attacker to the knowledge-soundness of $\mathcal{X}_{L2}$, we build an adversary $\mathcal{B} = (\mathcal{B}_{L1}, \mathcal{B}_{\mathcal{T}})$ that uses $\mathcal{A}$ as the inner-component to break the knowledge-soundness of $\mathcal{X}_{L1}$ or $\mathcal{X}_{\mathcal{T}}$.

Again, we define an auxiliary game $\mathbf{H}$ that is indistinguishable from the knowledge-soundness game and the probability that $\mathcal{A}$ wins in $\mathbf{H}$ is negligible.

Game $\mathbf{H}$: is the same as the game for knowledge-soundness of $\mathcal{X}_{L2}$, except that, the wining condition is modified by adding the following condition:

$$\mathsf{Cond}^* : \quad \mathcal{R}_{\mathcal{T}}(\gamma_v; v) = 0$$

We prove two following claims regarding $\mathbf{H}$.

- **Claim 1.** Game **H** is computationally indistinguishable from the game for knowledge-soundness (i.e., **G**), if $\mathcal{X}_{L1}$ is knowledge-sound.
- **Claim 2.** The probability of winning $\mathcal{A}$ in the game **H** is negligible if $\mathcal{X}_{\mathcal{T}}$ is knowledge-sound.

Proof of Claim 1: Assume the distinguisher $\mathcal{D}$ aims to distinguish **H** from **G**. The adversary $\mathcal{B}_{L1}$ (the attacker to the knowledge-soundness of $\mathcal{X}_{L1}$) simulates the games for the adversary $\mathcal{D}$ as follows.

- It receives the parameters for $\mathcal{X}_{L1}$, then runs the rest of the setup algorithm to get $\mathsf{pp}_{\mathcal{T}}$ and sends $\mathsf{pp}_{L2} = (\mathsf{pp}, \mathsf{pp}_{\mathcal{T}})$ to $\mathcal{D}$.
- when $\mathcal{D}$ responds with $x_{L2}, \pi_1, \pi_{\mathcal{T}}, \gamma_v$, the adversary $\mathcal{B}_{L1}$ runs the extractor $\mathcal{E}_{\mathcal{T}}$ to extract a witness v from $\pi_{\mathcal{T}}, \gamma_v$. Then, it forwards $(x = \xi^{-1}(x_{L2}, v), \pi_1)$ to its challenger and relays the challenge ρ.
- when $\mathcal{D}$ responds by π_2, the adversary $\mathcal{B}_{L1}$ outputs π_2.

We claim that the output $(x = \xi(v, x_{L2}))$ and (π_1, π_2) is an admissible output to break the knowledge-soundness of $\mathcal{X}_{L1}$. To see this, note that the only way $\mathcal{A}$ can distinguish the two games is when $(\mathsf{notCond}^*)$, that is, when $\mathcal{R}_{\mathcal{T}}(\gamma_v; v) = 1$ happens. Having $\mathcal{R}_{\mathcal{T}}(\gamma_v; v) = 1$ and the fact that verification $\mathcal{X}_{L2}$ passes result in passing the verification for $\mathcal{X}_{L1}$. While we have $\mathcal{R}_{L1}(v, x_{L2}; w) = 0$, this complete the proof for claim 1.

Proof of Claim 2: Let $\mathcal{A}$ be the attacker trying to win in the game **H**, we build the adversary $\mathcal{B}_{\mathcal{T}}$ that runs $\mathcal{A}$ as its inner-component to use its responses for breaking the knowledge-soundness of $\mathcal{X}_{\mathcal{T}}$, the simulation is as follows,

- It receives the parameters for $\mathcal{X}_{\mathcal{T}}$, by the fact that $\mathsf{pp} \subset \mathsf{pp}_{\mathcal{T}}$, it can find pp and sends $\mathsf{pp}_{L2} = (\mathsf{pp}, \mathsf{pp}_{\mathcal{T}})$ to $\mathcal{A}$.
- when $\mathcal{A}$ responds with $x_{L2}, \pi_1, \pi_{\mathcal{T}}, \gamma_v$, the adversary $\mathcal{B}_{\mathcal{T}}$ chooses the challenge ρ and sends it to $\mathcal{A}$.
- when $\mathcal{A}$ responds by π_2, it outputs $(\gamma_v; \pi_{\mathcal{T}})$.

Clearly, $(\gamma_v; \pi_{\mathcal{T}})$ is an admissible output for breaking the knowledge-soundness of $\mathcal{X}_{\mathcal{T}}$, since in this game we have $\mathcal{R}(\gamma_v; v) = 0$ while the verification passes.

I Proof of Theorem 3

Proof. Let $\mathcal{A}$ be the algebraic adversary attacking the knowledge-soundness of $\mathcal{X}_{\mathcal{T}}$, we show that the probability that it wins is negligible. The adversary receives $(g, g^{1/\sigma}, \mathsf{pp}_{Groth16}, \mathbf{L} = \mathsf{vk}_L^{\sigma})$ and outputs the proof $\pi \in \mathbb{G}_1$ and the public input $\gamma_v \in \mathbb{G}_1$. Define srs' as the element of srs in $\mathbb{G}_1$ that excludes **L**. It also outputs the vectors $\vec{a} = (\vec{a}_1, \vec{a}_2)$ and $\vec{b} = (\vec{b}_1, \vec{b}_2)$ as the LC-representations of π and γ_v (where $\vec{a}_2, \vec{b}_2$ are associated with the part **L** and $\vec{a}_1, \vec{b}_1$ are associated with the part srs'). The extractor $\mathcal{E}$ receives $\vec{a}, \vec{b}$. Note that LC-representations of π and

γ_v can be respectively written as follows (indeed we should use the $\mathsf{DL}(\mathsf{srs}')$ and $\mathsf{DL}(\mathsf{vk}_L)$)

$$a'_1 + \sigma a'_2 = \mathsf{srs}' \cdot \vec{a}_1 + \sigma \mathsf{vk}_L \vec{a}_2 \quad \text{and} \quad b'_1 + \sigma b'_2 = \mathsf{srs}' \cdot \vec{b}_1 + \sigma \mathsf{vk}_L \vec{b}_2$$

Since the verification passes, the adversary's response must satisfy the following equation (σ being the unknown here),

$$a'_1 + \sigma a'_2 = \sigma b'_1 + \sigma^2 b'_2 \tag{4}$$

There are two possible cases here, either the equation is satisfied independently of the choice of σ or it has a specific solution (w.r.t. σ as the unknown). We now discuss what will happen in each case.

1. The first case occurs if and only if $a'_1 = b'_2 = 0$ and $a'_2 = b'_1$, which then implies that $\vec{b}_2 = \vec{0}, \vec{a}_1 = \vec{0}$, $\vec{a}_2 = \vec{b}_{1,L} = \vec{a}^*$ and $\vec{b}_1 = (\vec{0}, \vec{b}_{1,L})$, for some $\vec{a}^*$. Thus, the extractor $\mathcal{E}$ has indeed extracted $\vec{a}^*$. But this can not pass the second condition in knowledge-soundness game which says $MSM(\mathsf{vk}_L, \vec{a}^*) \neq \gamma_v$ (by the CL-representation of γ_v, and the fact that $b_{1,L}$ is associated with the vk_L part). Thus, the probability that the extractor can output a witness satisfying the required condition is zero in this case.

2. In the second case, we can reduce knowledge-soundness to the DLog problem. Where the attacker $\mathcal{B}$ to the DLog-problem builds srs from its given challenge g_1, g_1^{σ}, runs the adversary $\mathcal{A}$ (attacker to the knowledge-soundness), then as explained in the current case it can solve Eq. (4) to find σ.

Therefore, we have shown that the probability that Eq. (4) has a specific solution is negligible, which means the case (1) would happen with overwhelming probability, while in this case the probability that expected extractor $\mathcal{E}$ exists is zero. Putting together the knowledge-error is at most ϵ_{DL} where ϵ_{DL} is the probability of breaking DLog-assumption.

J Fiat-Shamir transform in the so-far digest model

In the way that we instantiate the Fiat-Shamir transform, we only hash the current message and the previous randomness to obtain the new randomness. We call this model as the so-far digest, as opposed to the so-far transcript which is usually used in the original version of Fiat-Shamir transform. We argue that there is no concrete difference between the two approaches, as ultimately each Fiat-Shamir randomness depends on the previous transcript. Indeed, each randomness is the digest of the so-far transcript (thus, the name "so-far digest").

Motivation. In real applications, the random oracle is replaced with a hash function. And numerous hash functions are *updatable*, meaning that the hash of the message

$$M = M' \| \text{new block of message}$$

can be built from the new block of the message and a short input, the *hash state* after hashing M'. This means for an updatable hash two models so-far transcript and so-far digest (with the same hash for all the rounds) are, in practice, roughly equivalent in terms of performances. The reason we prefer *the so-far digest* model is mainly due to our one-round GKR. If we had used the *so-far transcript model*, the verifier would still have to hash the entire transcript from scratch in the second round (public parameters, public inputs, past prover's messages etc...). Indeed, in that scenario, the random-coin of the verifier for the first round would count as a *message* which is part of the transcript and cannot be used as a *previous hash state*. As a result, we would require the prover to hash a potentially very long string of input within a SNARK circuit which is what we are trying to avoid in this work.

J.1 From so-far Transcript model to so-far digest model

Here, we formally define the so-far digest model which provide us with a more efficient implementation of Fiat-Shamir transform in the circuit. Then, we prove that if a protocol is sound in so-far transcript random oracle model, it is also sound in the so-far digest random oracle model.

Definition 18 (So-Far Digest Model). *Let $\mathcal{P}$ be an FS-transform of a public-coin interactive protocol in the random oracle model. Let also m_i and h_i be respectively the prover massage and the output of the FS-hash in the round i. In the so-far digest model, we have $h_i = H_{\mathcal{FS}}(m_i, h_{i-1})$ for $i > 1$, and $h_1 = H_{\mathcal{FS}}(m_1, \mathsf{pp}, x)$ where pp stands for the public parameters and x is the public input of the verifier.*

Note that in the so-far transcript model we simply have

$$h_i = H_{\mathcal{FS}}(\text{so-far Transcript}_i)$$

and *so-far transcript* is whatever the verifier has received so far during the protocol execution (this includes all the public parameters, the public input of the verifier and all the messages of the prover).

Theorem 4. *If $\mathcal{P}$ (defined above) is sound in the so-far transcript model with soundness error ϵ_{tr}, then it is also sound in the so-far digest model with soundness-error $t^{-1}.\epsilon_{tr}$ where $t = (1-1/2^{\lambda_H})$ and λ_H is the length random oracle's response size (in bits).*

Proof. Let $\mathcal{A}$ and $\mathcal{B}$ be the attacker respectively to the soundness of P in the so-far transcript model and in the so-far digest model. The attacker $\mathcal{A}$ simulates the soundness game for the adversary $\mathcal{B}$ as follows,

- let $\mathcal{B}$ outputs the message m_i in round i. If later it sends a RO-query on (m_i, h_k) with $k = i - 1$, the adversary $\mathcal{A}$ simply updates the transcript of its game and forwards m_i to its challenger (as its response for the round i) where it also sends a RO-query to $H_{\mathcal{FS}}$ on the updated transcript. It receives a response h_i which would be relayed to $\mathcal{B}$.

– It responds with the random values for other cases, and repeat the same response for the repeated queries.

In the case of the "Bad Event", defined in the following, the adversary $\mathcal{A}$ would abort and otherwise it outputs the output of $\mathcal{B}$.

Bad Event. is when $\mathcal{B}$ sends a RO-query on (m, h) to $H_{\mathcal{FS}}$, for m that has not been the response for any round, later during the i-th round it sends $m_i = m$ as its message, while we also have $h_{i-1} = h$. This means RO-query (m, h) was responded by a random value while at round i it is required to be responded based on a RO-query of $\mathcal{A}$ (conflicting the simulation). This event happens with negligible probability since $h = h_{i-1}$ happens only with $1/2^{\lambda_H}$ where λ_H is the output-length of the FS-hash function.

Clearly, the adversary $\mathcal{A}$ can win its game if it has not aborted (bad event hash not happened) and $\mathcal{B}$ has won its game.