Stacking Sigmas:
A Framework to Compose Σ-Protocols for Disjunctions

Aarushi Goel[1], Matthew Green[1], Mathias Hall-Andersen[2], and Gabriel Kaptchuk[3]

[1]Johns Hopkins University, {aarushig,mgreen}@cs.jhu.edu
[2]Aarhus University, ma@cs.au.dk
[3]Boston University, kaptchuk@bu.edu

Abstract

Zero-Knowledge (ZK) Proofs for disjunctive statements have been a focus of a long line of research. Classical results such as Cramer *et al.* [CRYPTO'94] and Abe *et al.* [AC'02] design generic compilers that transform certain classes of ZK proofs into ZK proofs for disjunctive statements. However, communication complexity of the resulting protocols in these results ends up being proportional to the complexity of proving all clauses in the disjunction. More recently, Heath *et al.* [EC'20] exploited special properties of garbled circuits to construct efficient ZK proofs for disjunctions, where the proof size is only proportional to the length of the largest clause in the disjunction. However, these techniques do not appear to generalize beyond garbled circuits.

In this work, we focus on achieving the best of both worlds. We design a *general framework* that compiles a large class of unmodified Σ-protocols, each for an individual statement, into a new Σ-protocol that proves a disjunction of these statements. Our framework can be used both when each clause is proved with the same Σ-protocol and when different Σ-protocols are used for different clauses. The resulting Σ-protocol is concretely efficient and has communication complexity proportional to the communication required by the largest clause, with additive terms that are only logarithmic in the number of clauses.

We show that our compiler can be applied to many well-known Σ-protocols, including classical protocols (*e.g.* Schnorr [JC'91] and Guillou-Quisquater [CRYPTO'88]) and modern MPC-in-the-head protocols such as the recent work of Katz, Kolesnikov and Wang [CCS'18] and the Ligero protocol of Ames *et al.* [CCS'17]. Finally, since all of the protocols in our class can be made non-interactive in the random oracle model using the Fiat-Shamir transform, our result yields the first generic non-interactive zero-knowledge protocol for disjunctions where the communication only depends on the size of the largest clause.

Contents

1 Introduction **3**
1.1 Our Contributions. . . . 4

2 Related Work **5**

3 Technical Overview **6**

4 Preliminaries **11**
4.1 Notation . . . 11
4.2 $\mathbf{\Sigma}$-Protocols . . . 11
4.3 Secure Multiparty Computation . . . 12

5 Partially-Binding Vector Commitments **12**
5.1 Relation To Similar Notions . . . 14
5.2 Partially-Binding Vector Commitments from Discrete Log . . . 14
5.3 Generic Construction of 1-of-2^q Partially-Binding Vector Commitment. . . . 16

6 Stackable Σ-Protocols **18**
6.1 Properties of Stackable Σ-Protocols. . . . 18
6.1.1 Cheat Property: "Extended" Honest Verifier Zero-Knowledge. . . . 18
6.1.2 Re-use Property: Recyclable Third Round Messages. . . . 20
6.1.3 Stackability . . . 21
6.2 Classical Examples of Stackable Σ-Protocols . . . 21
6.3 Examples of Stackable "MPC-in-the-Head" $\mathbf{\Sigma}$-Protocols . . . 22
6.4 Well-Behaved Simulators . . . 26

7 Self-Stacking: Disjunctions With The Same Protocol **27**
7.1 Self Stacking for Instances in Multiple Languages . . . 28

8 Cross-Stacking: Disjunctions with Different Protocols **30**
8.1 Cross Simulatability . . . 30
8.2 Cross-Stacking from Cross Simulatability . . . 34

9 k-out-of-ℓ Proofs of Partial Knowledge **34**

10 Measuring Concrete Efficiency **35**

A Blum87 is Stackable: Proof of Lemma 2 **41**

B Well-Behaved Simulators: Proof of Lemma 5 **42**

C Security Proof for Cross-Stacking Compiler (Theorem 6) **43**

D Overview of [KKW18] and Proof of Lemma 3 **44**
D.1 [KKW18] is Stackable: Proof of Lemma 3 . . . 47

E Overview of Ligero and Proof of Lemma 4 **47**
E.1 Ligero is Stackable: Proof of Lemma 4 . . . 50

F k-out-of-ℓ Proofs Of Partial Knowledge (Old Construction) **50**

G Optimized Partially Binding Vector Commitments from RO **54**

1 Introduction

Zero-knowledge proofs and arguments [GMR85] are cryptographic protocols that enable a prover to convince the verifier of the validity of an NP statement without revealing the corresponding witness. These protocols, along with proof of knowledge variants, have now become critical in the construction of larger cryptographic protocols and systems. Since classical results established feasibility of such proofs for all NP languages [GMW86], significant effort has gone into making zero-knowledge proofs more practically efficient *e.g.* [JKO13, BCTV14, Gro16, KKW18, BBB+18, BCR+19, HK20b], resulting in concretely efficient zero-knowledge protocols that are now being used in practice [BCG+14, Zav20, se19].

Zero-knowledge for Disjunctive Statements. There is a long history of developing zero-knowledge techniques for *disjunctive statements* [CDS94, AOS02, GMY03]. Disjunctive statements comprise of several *clauses* that are composed together with a logical "OR." These statements also include conditional clauses, *i.e.* clauses that would only be relevant if some condition on the statement is met. The witness for such statements consists of a witness for one of the clauses (also called the *active* clause), along with the index identifying the active clause. Disjunctive statements occur commonly in practice, making them an important target for proof optimizations. For example, disjunctive proofs are often also used to give the prover some degree of privacy, as a verifier cannot determine which clause is being satisfied. Use cases include membership proofs (*e.g.* ring signatures [RST01]), proving the existence of bugs in a large codebase (as explored in [HK20b]), and proving the correct execution of a processor, which is typically composed of many possible instructions, only one of which is executed at a time [BCG+13].

An exciting line of recent work has emerged that reduces the communication complexity for proving disjunctive statements to the size of the largest clause in the disjunction [Kol18, HK20b]. While succinct proof techniques exist [Gro10, GGPR13, BCTV14, Gro16], known constructions are plagued by very slow proving times and often require strong assumptions, sometimes including trusted setup. These recent works accept larger proofs in order to get significantly faster proving times and more reasonable assumptions — while still reducing the size of proofs significantly. Intuitively, the authors leverage the observation that a prover only needs to honestly execute the parts of a disjunctive statement that pertain to their witness. Using this observation, these protocols modify existing proof techniques, embedding communication-efficient ways to "cheat" for the inactive clauses of the disjunctive statement. We refer to these techniques as *stacking* techniques, borrowing the term from the work of Heath and Kolesnikov [HK20b].

Although these protocols achieve impressive results, designing stacking techniques requires significant manual effort. Each existing protocol requires the development of a novel technique that reduces the communication complexity of a specific base protocol. For instance, Heath and Kolesnikov [HK20b] observe that garbled circuit tables can be additively *stacked* (thus the name), allowing the prover in [JKO13] to *un-stack* efficiently, leveraging the topolgy hiding property of garbling. Techniques like these are tailored to optimize the communication complexity of a particular underlying protocol, and do not appear to generalize well to large families of protocols. In contrast, classical results [CDS94, AOS02] succeed in designing a generic compiler that tranforms a large familily of zero-knowledge proof systems into proofs for disjunction, but fall short of reducing the size of the resulting proof.

In this work, we take a more general approach towards reducing the communication complexity of zero-knowledge protocols for disjunctive statements. Rather than reduce the communication complexity of a specific zero-knowledge protocol, we investigate *generic* stacking techniques for an important family of zero-knowledge protocols — three round public coin proofs of knowledge, popularly known as Σ-protocols. Specifically, we ask the following question:

Can we design a generic compiler that stacks any Σ-protocol without modification?

We take significant steps towards answering this question in the affirmative. While we do not demonstrate a technique for stacking all Σ-protocols, we present a compiler that *stacks* many natural Σ-protocols, including many of practical importance. We focus our attention on Σ-protocols because of their widespread use and because they can be made non-interactive in the random oracle model using the Fiat-Shamir transform [FS87]. However we expect that the techniques can easily be generalized to public-coin protocols with more rounds.

Benefits of a Generic Stacking Compiler. There are several significant benefits of developing generic stacking compilers, rather than developing bespoke protocols that support stacking. First, automatically compiling multiple Σ-protocols into ones supporting stacking removes the significant manual effort required to modify existing techniques. Moreover, newly developed Σ-protocols can be used to produce stacked proofs immediately, significantly streamlining the deployment process. A second, but perhaps even more practically consequential, benefit of generic compilers is that protocol designers are empowered to tailor their choice of Σ-protocol to their application — without considering if there are known stacking techniques for that particular Σ-protocol. Specifically, the protocol designer can select a proof technique that fits with the natural representation of the relevant statement (*e.g.* Boolean circuit, arithmetic circuit, linear forms or any other algebraic structure). Without a generic stacking compiler, a protocol designer interested in reducing the communication complexity of disjunctive proofs might be forced to apply some expensive NP reduction to encode the statement in a stacking-friendly way. This is particularly relevant because modern Σ-protocols often require that relations are phrased in a very specific manner, *e.g.* Ligero [AHIV17] requires arithmetic circuits over a large, finite field, while known stacking techniques [HK20b] focus on Boolean circuits.

A common concern with applying protocol compilers is that they trade generality for efficiency (*e.g.* NP reductions). However, we note that the compiler that we develop in this work is extremely concretely efficient, overcoming this common limitation. For instance, naïvely applying our protocol to the classical Schnorr identification protocol and applying the Fiat-Shamir [FS87] heurestic yields a ring signature construction with signatures of length $2\lambda \cdot (2 + 2\log(\ell))$ bits, where λ is the security parameter and ℓ is the ring size; this is actually smaller than modern ring signatures from similar assumptions [BCC$^+$15, ACF20] without requiring significant optimization. [1]

1.1 Our Contributions.

In this work, we give a generic treatment for minimizing the communication complexity of Σ-protocols for disjunctive statements. In particular, we identify some "special properties" of Σ-protocol that make them amenable to "stacking." We refer to protocols that satisfy these properties as *stackable* protocols. Then we present a framework for compiling any stackable Σ-protocols for independent statements into a new, communication-efficient Σ-protocol for the disjunction of those statements. Our framework only requires oracle access to the prover, verifier and simulator algorithms of the underlying Σ-protocols. We present our results in two-steps:

Self-Stacking Compiler. First, we present our basic compiler, which we call a "self-stacking" compiler. This compiler composes several instances of the *same* Σ-protocol, corresponding to a particular language into a disjunctive proof. The resulting protocol has communication complexity proportional to the communication complexity of a single instance of the underlying protocol. Specifically, we prove the following theorem:

Informal Theorem 1 (Self-Stacking)**.** *Let* Π *be a stackable* Σ*-protocol for an NP language* $\mathcal{L}$ *that has communication complexity* $\mathtt{CC}(\Pi)$*. There exists is a* Σ*-protocol for the language* $(x_1 \in \mathcal{L}) \vee \ldots \vee (x_\ell \in \mathcal{L})$*, with communication complexity* $O(\mathtt{CC}(\Pi) + \lambda \log(\ell))$*, where* λ *is the computational security parameter.*

Cross-stacking. We then extend the self-stacking compiler to support stacking *different* Σ-protocols for different languages. The communication complexity of the resulting protocol is a function of the largest clause in the disjunction and the similarity between the Σ-protocols being stacked. Let $f_{\mathtt{CC}}$ be a function that determines this dependence. For instance, if we compose the same Σ-protocol but corresponding to different languages, then the output of $f_{\mathtt{CC}}$ will likely be the same as that of a single instance of that protocol for the language with the largest relation function. However, if we compose Σ-protocols that are very different from each other, then the output of $f_{\mathtt{CC}}$ will likely be larger. We prove the following theorem:

Informal Theorem 2 (Cross-Stacking)**.** *For each* $i \in [\ell]$*, let* Π_i *be a stackable* Σ*-protocol for an NP language* $\mathcal{L}_i$ *There exists is a* Σ*-protocol for the language* $(x_1 \in \mathcal{L}_1) \vee \ldots \vee (x_\ell \in \mathcal{L}_\ell)$*, with communication complexity* $O(f_{\mathtt{CC}}(\{\Pi_i\}_{i\in[\ell]}) + \lambda \log(\ell))$*.*

[1] Although concrete efficiency is a central element of our work, applying our compiler to applications is not our focus. The details of this ring signature construction can be found in Section 10.

Examples of Stackable Σ-protocols. We show many concrete examples of Σ-protocols that are stackable. Specifically, we look at classical protocols like Schnorr [Sch90], Guillio-Quisquater [GQ90] and Blum [Blu87], and modern MPC-in-the-head protocols like KKW [KKW18] and Ligero [AHIV17]. Previously it was not known how to prove disjunction over these Σ-protocols with sublinear communication in the number of clauses. When applied to these Σ-protocols, our compiler yields a Σ-protocol which can can made non-interactive in the random oracle model using the Fiat-Shamir heurestic. For example, when instantiated with Ligero our compiler yields a concretely efficient Σ-protocol for disjunction over ℓ different circuits of size $|C|$ each, with communication $O(\sqrt{|C|} + \lambda \log \ell)$. Additionally, we explore how to apply our cross-stacking compiler to stack different stackable Σ-protocols with one another (*e.g.* stacking a KKW proof for one relation with a Ligero proof for another relation).

Partially-binding non-interactive vector commitments. Central to our compiler is a new variation of commitments called partially-binding non-interactive vector commitment schemes. These schemes allow a committer to commit to a vector of values and equivocate on a subset of the elements in that vector, the positions of which are determined during commitment and are kept hidden. We show how such commitments can be constructed from the discrete log assumption.

Extensions and Implementation Considerations. We finish by discussing extensions of our work and concrete optimizations that improve the efficiency of our compiler when implemented in practice. Specifically, we consider generalizing our work to k-out-of-ℓ proofs of partial knowledge, *i.e.* the threshold analog of disjunctions. We give a version of our compiler that works for these threshold statements. Additionally, we demonstrate the efficiency of our compiler by presenting concrete proof sizes when our compiler is applied to both a disjunction of KKW and Schnorr signatures.

Future Work. In this work we focus on Σ-protocols for ease of explication and to capture a wide class of interesting protocols, however it should be possible to extend our techniques to zero-knowledge proofs with more rounds using suitable generalizations.

2 Related Work

Proofs of partial knowledge. The classic work of Cramer et. al. [CDS94] shows how to compile a secret-sharing scheme and Σ-protocols for the (possibly distinct) relations $\mathcal{R}_1, \ldots, \mathcal{R}_\ell$ into a new Σ-protocols (without additional assumptions) for the t-threshold "partial knowledge" relation $\mathcal{R}_{t,(\mathcal{R}_1,\ldots,\mathcal{R}_\ell)}(x, w) \coloneqq |\{\mathcal{R}_i(x_i, w_i) = 1\}| \geq t$. The communication of the resulting Σ-protocol is $|\pi| = O(\ell)$. Abe et. al. [AOS02] suggested an alternative approach to creating 1-of-n proofs in the non-interactive context of ring signatures. Specifically, the prover (starting with the active clause) hashes the first round message of the i^{th} clause to generate the challenge for the $(i+1)^{\text{th}}$ clause; for each inactive clause, the prover uses a simulator to complete the transcript with respect to the generated challenge. The resulting signature contains a third round message for each clause, making it linear in ℓ. Because their approach requires generating the third round message of the active clause *after* simulating the inactive clauses, it is not clear how to generalize their techniques to re-use messages. Groth and Kohlweiss [GK15] constructed a zero-knowledge proof of partial knowledge for the "discrete log" relation i.e. $\mathcal{R}_1 = \ldots = \mathcal{R}_\ell = \mathcal{R}_{\text{dlog}} \coloneqq x \stackrel{?}{=} g^w$ with threshold $t = 1$ and communication $|\pi| = O(\log \ell)$. Later work by Attema, Cramer and Fehr [ACF20] obtains proofs of partial knowledge for $\mathcal{R}_{\text{dlog}}$ with any threshold t and $|\pi| = O(\log \ell)$ communication, by applying compressed Σ-protocol theory [AC20]. Work by Jivanyan and Manikonyan [JM20] reduces the computational overhead of similar proofs from $O(\ell \log \ell)$ to $O(\ell)$ at the cost of communication. Unlike these earlier/concurrent '$O(\log n)$ works', we considers a much broader class of Σ-protocols and deploy fundamentally different techniques.

Online/offline OR composition of Σ-protocols. Ciampi *et al.* [CPS+16] extended the 'proof of partial knowledge' work by Cramer *et al.* to enable specifying instances in the disjunction in the third round. This is attained by constructing a (k, n)-equivocal commitment scheme from Σ-protocols and the original Cramer. et. al compiler [CDS94]. Careful analysis shows that despite the prover being able to adaptively choose

instances the transformation is sound. The goal of Ciampi *et al.* is very different and does not consider communication saving, but our work makes use of similar (k, n)-equivocal commitments (called 'partially-binding commitments' here) to obtain communication savings rather than delayed instance specification.

Stacked Garbling. Work by Heath and Kolesnikov [HK20b], extends the works of Jawurek et. al. [JKO13] and Frederiksen et. al. [FNO15] to obtain efficient interactive zero-knowledge proofs over disjunctive statements (Boolean circuits) This is done by having the garbler garble each clause separately then "stacking" the garbled circuits by XORing them together. The stacked result is sent to the verifier, who obliviously retrieves the garbling randomness for all but one of the garbled circuits and reconstructs the remaining garbling circuit. Subsequent work [HK20a] by the same authors, extended similar stacking techniques to enable 2PC with communication saving for circuits with disjunctions, without the need for a separate output selection protocol as in [Kol18].

Mac'n'Cheese. Concurrent work by Baum et. al [BMRS20] introduces an abstraction dubbed LOVe ('Interactive Protocols with Linear Oracle Verification') and obtains 'free nested disjunctions' for this class of interactive zero-knowledge proofs. They give a concretely efficient constant-round instantiation of a LOVe for satisfiability of a arithmetic circuits over sufficiently large fields in the RO model. Since soundness relies on the prover maintaining linear MACs (message authentication codes) established using VOLE (Vector Oblivious Linear Evaluation) under a verifier's secret key, it is not obvious how to make this protocol non-interactive.

3 Technical Overview

In this section, we give a detailed overview of the techniques that we use to design a generic framework to achieve communication-efficient disjunctions of Σ-protocols without requiring non-trivial[2] changes to the underlying Σ-protocols. Throughout this work, we consider a disjunction of ℓ *clauses*, one (or more) of which are *active*, meaning that the prover holds a witness satisfying the relation encoded into those clauses. For the majority of this technical overview, we focus on the simpler case where the same Σ-protocol is used for each clause. We will then extend our ideas to cover heterogeneous Σ-protocols.

Recall that Σ-protocols are three-round, public-coin zero-knowledge protocols, where the prover sends the first message. In the second round, the verifier sends a random "challenge" message to the prover, that only depends on the random coins of the the verifier. Finally, in the third round, the prover responds with a message based on this challenge. Based on this transcript the verifier then decides whether to accept or reject the proof.

We start by considering the approaches taken by recent works focusing on privacy-preserving protocols for disjunctive statements, *e.g.* [HK20b]. We observe that the "stacking" techniques used in all these works can be broadly classified as taking a *cheat and re-use approach.* In particular, all of these works show how some existing protocols can be modified to allow the parties to "cheat" on the inactive clauses — *i.e.* only executing the active clause honestly — and "re-using" the single honestly-computed transcript to mimic a fake computation of the inactive clauses. Critically, this is done while ensuring that the verifier cannot distinguish the honest execution of the active clause from the fake executions of the inactive clauses.

Our Approach. In this work we extend the *cheat and re-use* approach to design a framework for compiling Σ-protocols into a communication-efficient Σ-protocol for disjunctive statements without requiring modification of the underlying protocols. Specifically, we are interested in reducing the number of *third round messages* that a prover must send to the verifier, since the third round message is typically the longest message in the protocol. Intuition extracted from prior work leads us to a natural high-level template for

[2]We assume that basic, practice-oriented optimizations have already been applied to the Σ-protocols in question. For instance, we assume that only the minimum amount of information is sent during the third round of protocol. Hereafter, we will ignore these trivial modifications and simply say "without requiring modification." Note that these modifications truly are trivial: the parties only need to repeat existing parts of the transcript in other rounds. We discuss this in the context of MPC-in-the-head protocols in Section 6.

achieving this goal: *Run individual instances of Σ-protocols (one-for each clause in the disjunction) in parallel, such that only one of these instances (the one corresponding to the active clause) is honestly executed, and the remaining instances re-use parts of this honest instance.*

There are two primary challenges we must overcome to turn this rough outline into a concrete protocol: (1) how can the prover cheat on the inactive clauses? and (2) what parts of an honest Σ-protocol transcript can be safely re-used (without revealing the active clause)? We now discuss these challenges, and the techniques we use to overcome them, in more detail.

Challenge 1: How will the prover cheat on inactive clauses? Since the prover does not have a witness for the inactive clauses, the prover can cheat by creating accepting transcripts for the inactive clauses using the simulator(s) of the underlying Σ-protocols. The traditional method (*e.g.* [CDS94] for disjunctive Schnorr proofs) requires the prover to start the protocol by randomly selecting a challenge for each inactive clause and simulating a transcript with respect to that challenge. In the third round, the prover completes the transcript for each clause and demonstrates that it could only have selected the challenges for all-but-one of the clauses. This approach, however, inherently requires sending many third round messages, which will make it difficult to re-use material across clauses (discussed in more detail below). Similarly, alternative classical approaches for composing Σ-protocols for disjunctives, like that of Abe et al. [AOS02], also require sending a distinct third round message for each clause. As such, we require a new approach for cheating on the inactive clauses.

Our first idea is to defer the selection of first round messages for the inactive clauses until after the verifier sends the challenge (*i.e* in the third round of the compiled protocol), while requiring that the prover select a first round message honestly for the active clause (*i.e* in the first round of the compiled protocol). To do this, we introduce a new notion called *non-interactive, partially-binding vector commitments.*[3] These commitments allow the committer to commit to a vector of values and equivocate on a hidden subset of the entries in the vector later on. For instance, a 1-out-of-ℓ binding commitment allows the committer to commit a vector of ℓ values such that that one of the vector positions (chosen when the commitment is computed) is binding, while allowing the committer to modify/equivocate the remaining positions at the time of opening. For a disjunction with ℓ clauses, we can now use this primitive to ensure that the prover computes an honest transcript for at least one of the Σ-protocol instances as follows:

- **Round 1:** The prover computes an honest first round message for the Σ-protocol corresponding to the active clause. It commits to this message in the binding location of a 1-out-of-ℓ binding commitment, along with $\ell - 1$ garbage values, and sends the commitment to the verifier.
- **Round 2:** The verifier sends a challenge message for the ℓ instances.
- **Round 3:** The prover honestly computes a third round message for the active clause and then simulates first and third round messages for the remaining $\ell - 1$ clauses. It equivocates the commitment with these updated first round messages, and sends an opening of this commitment along with all the ℓ third round messages to the verifier.

While this is sufficient for soundness, we need an additional property from these partially-binding vector commitments to ensure zero-knowledge. In particular, in order to prevent the verifier from learning the index of the active clause, we require these partially-binding commitments to not leak information about the binding vector position. We formalize these properties in terms of a more general t-out-of-ℓ binding vector commitment scheme, which may be of independent interest, and we provide a practical construction based on the discrete log assumption.[4]

Challenge 2: How will the prover re-use the active transcript? The above approach overcomes the first challenge, but doesn't achieve our goal of reducing the communication complexity of the compiled Σ-protocol. Next, we need to find a way to somehow re-use the honest transcript of the active clause. Our key insight is that for many natural Σ-protocols, it is possible to simulate *with respect to a specific third*

[3] A similar notion for interactive commitments was introduced in [CPS+16].

[4] We also explore a construction that is half the size and leverages random oracles in Appendix G.

round message. That is, it is often easy to simulate an accepting transcript for a given challenge and third round message. This allows the prover to create a transcript for the inactive clauses that share the third round message of the active clause. In order for this compilation approach to work, Σ-protocols must satisfy the following properties (stated here informally):

– *Simulation With Respect To A Specific Third Round Message:* To re-use the active transcript, the prover simulates *with respect to the third round message of the active transcript.* This allows the prover to send a single third round message that can be re-used across all the clauses. More formally, we require that the Σ-protocol have a simulator that can reverse-compute an appropriate first round message to complete the accepting transcript for any given third round message and challenge. While not possible for all Σ-protocols, simulating in this way—i.e., by first selecting a third round message and then "reverse engineering" the appropriate first round message—is actually a common simulation strategy, and therefore possible with most natural Σ-protocols. In order to get communication complexity that only has a logarithmic dependence on the number of clauses, we additionally require this simulator to be deterministic.[5] We formalize this property in Section 6.

– *Recyclable Third Round Messages:* To re-use third round messages in this way, the distribution of these third round messages must be the same. Otherwise, simulating the inactive clauses would fail and the verifier could detect the active clause used to produce the third round message. Thus, we require that the distribution of third round messages in the Σ-protocol be the same across all statements of interest. We formalize this property in Section 6.

An mentioned before, most natural Σ-protocols satisfy both these properties and we refer to such protocols as *stackable* Σ*-protocols*. We can compile such Σ-protocols into a communication-efficient Σ-protocol for disjunctions, where the communication only depends on the size of one of the clauses, as follows: Rounds 1 and 2 remain the same as in the protocol sketch above. In the third round, the prover first computes a third round message for the active clause. It then simulates first round messages for the remaining clauses based on the active clause's third round message and the challenge messages. As before, it equivocates the commitment with these updated first round messages.[6] While this allows us to compress the third round messages, we still need to send a vector commitment of the first round messages. In order to get communication complexity that does not depend on the size of all first round messages, the size of this vector commitment should be independent of the size of the values committed. Note that this is easy to achieve using a hash function.

Summary of our Stacking Compiler. Having outlined our main techniques, we now present a detailed description of our compiler for 2 clauses, as depicted in Figure 1 (similar ideas extend for more than 2 clauses). The right (unshaded) box represents the active clause and the left (shaded) box represents the inactive clause. Each of the following numbered steps refer to a correspondingly numbered arrow in the figure: (1) The prover runs the first round message algorithm of the active clause to produce a first round message a_2. (2) The prover uses the 1-of-2 binding commitment scheme to commit to the vector $\mathbf{v} = (0, a_2)$. (3) The resulting commitment constitutes the compiled first round message a'. (4) The challenge c' is created by the verifier. (5) The prover generates the third round message z for the active clause using the first round message a_2, the challenge c', and the witness w. (6) The prover then uses the simulator for the inactive clause on the challenge c' and the honestly generated third round message z to generate a valid first round message for the inactive clause a_1. (7) The prover equivocates on the contents of the commitment a' – replacing 0 with the simulated first round message a_1. The result is randomness r' that can be used to open commitment a' to the vector $\mathbf{v}' = (a_1, a_2)$. (8) The compiled third round message consists of honestly generated third round message z, the randomness r' of the equivocated commitment, and the two first round

[5] We elaborate on the importance of this additional property in the technical sections.

[6] If the simulator computes the first round messages deterministically, then the prover only needs to reveal the randomness used in the commitment in the third round, along with the common third round message to the verifier. Given the third round message, the verifier can compute the first round messages on its own and check if the commitment was valid and that the transcripts verify.

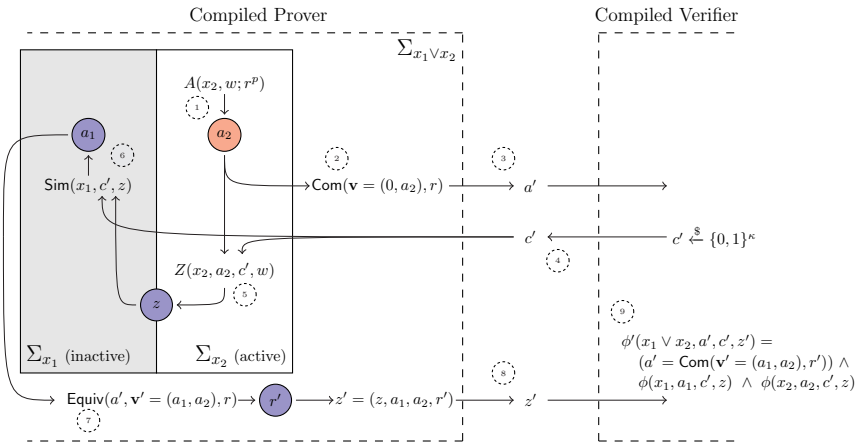

Figure 1: High level overview of our compiler applied to a Σ-protocol $\Sigma = (A, C, Z, \phi)$ over statements x_1 and x_2. Several details have been omitted or changed to illustrate the core ideas more simply. The red circle contains a value used in the first round, while purple circles contain values used in the third round. We include a_1 and a_2 in the third round message for clarity; in the real protocol, the verifier will be able to deterministically recompute these values on their own.

messages a_1, a_2.[7] (9) The verifier then verifies the proof by ensuring that each transcript is accepting and that the first round messages constitute a valid opening to the commitment a'.

Complexity Analysis: Communication in the first round only consists of the commitment, which we show can be realized in $O(\ell\lambda)$ bits, where λ is the security parameter. In the last round, the prover sends one third round message of the underlying Σ-protocol that depends on the size of one of the clauses[8] and ℓ first round messages of the underlying Σ-protocol. Thus, naïvely applying our compiler results in a protocol with communication complexity $O(\mathtt{CC}(\Sigma) + \ell \cdot \lambda)$, where $\mathtt{CC}(\Sigma)$ is the communication complexity of the underlying stackable Σ-protocol, when executed for the largest clause. In the technical sections, we show that the resulting protocol is itself "stackable", it can be recursively compiled. This reduces the communication complexity to $O(\mathtt{CC}(\Sigma) + \log(\ell) \cdot \lambda)$.

Stackable Σ-Protocols. While not all Σ-protocols are able to satisfy the first two properties that we require, we show that many natural Σ-protocols like Schnorr [Sch90], and Guillio-Quisquater [GQ90] satisfy these properties. We also show that more recent state-of-the-art protocols in MPC-in-the-head paradigm [IKOS07] like KKW [KKW18] and Ligero [AHIV17] have these properties. We formalize the notion of "$\mathcal{F}$-universally simulatable MPC protocols", which produce stackable Σ-protocols when compiled using MPC-in-the-head [IKOS07]. This formalization is highly non-trivial and requires paying careful attention to the distribution of MPC-in-the-head transcripts. Our key observation is that transcripts generated when executing one circuit can often be seamlessly *reinterpreted* as though they were generated for another circuit (usually of similar size). We refer the reader to Section 6 for more details on stackable Σ-protocols.

Stacking Different Σ-Protocols. The compiler presented above allows stacking transcripts for a single Σ-protocol, with a single associated NP language, evaluated over different statements e.g., $(x_1 \in \mathcal{L}) \vee \ldots \vee (x_\ell \in \mathcal{L})$. This is quite limiting and does not allow a protocol designer to select the optimal Σ-protocol for each clause in a disjunction. As such, we explore extending our compiler to support stacking *different* Σ-protocols with different associated NP languages, *i.e.* $(x_1 \in \mathcal{L}_1) \vee (x_2 \in \mathcal{L}_2) \vee \ldots \vee (x_\ell \in \mathcal{L}_\ell)$.

We start by noting that it is possible to create a "meta-language" to cover multiple languages of interest, and thereby generalize our previous compiler in a straightforward way. For instance, one could create a

[7] In the compiler presented in the main body, a_1 and a_2 are omitted from the third round message and the verifier recomputes them from z and c' directly. We make this simplification in the exposition to avoid introducing more notation.

[8] We can assume w.l.o.g. that all clauses have the same size. This can be done by appropriately padding the smaller clauses.

language $\mathcal{L}$ with an associated relation function that embeds the relation functions for $\mathcal{L}_1, \ldots, \mathcal{L}_\ell$, making $\mathcal{L}$ some form of circuit satisfiability language. A single Σ-protocol could then be used to cover all these languages. Unfortunately, this approach — intuitively equivalent to creating zero-knowledge protocols for all NP complete problems by reducing to a single problem — will often result in high concrete overheads. In rare cases, however, it may be practically efficient; if the languages $\mathcal{L}_1, \ldots, \mathcal{L}_\ell$ are all circuit satisfiability for circuits with the same multiplicative complexity, finding an efficient representation might be easy.

This "meta-language" approach still requires the use of a single Σ-protocol. It would be preferable to allow "cross-stacking," or using different Σ-protocols for each clause in the disjunction.[9] The key impediment to applying our self-stacking compiler to different Σ-protocols is that the distribution of third round messages between two different Σ-protocols may be very different. For example, a statement with three clauses may be composed of one Σ-protocol defined over a large, finite field, another operating over a boolean circuit, and a third that is constructed from elements of a discrete logarithm group. Thus, attempting to use the simulator for one Σ-protocol with respect to the third round message of another might result in a domain error; there may be no set of accepting transcripts for the Σ-protocols that share a third round message. As re-using third round messages is the way we reduce communication complexity, this dissimilarity might appear to be insurmountable.

To accommodate these differences, we observe that the extent to which a set of Σ-protocols can be stacked is a function of the similarity of their third round messages. In the self-stacking compiler, these distributions were exactly the same, resulting in a "perfect stacking." With different Σ-protocols, the prover may only be able to re-use a *part* of the third round message when simulating for another Σ-protocol, leading to a "partial stacking." We note, however, that the distributions of common Σ-protocols tend to be quite similar — particularly when seen as an unstructured string of bits. For instance, transcript containing points on Curve25519 encoded using Elligator [BHKL13], elements of $\mathbb{Z}_{2^{16}}$, and field elements in $\mathbb{F}_{2^{64}}$ will all appear to be random bitstrings when viewed without structure, and will be indistinguishable (assuming correct padding). These random bitstrings can then be partitioned and interpreted, as needed, by each simulator.

More formally, stacking different Σ-protocols requires an efficient, invertible mapping from each third round message space into some shared distribution $\mathcal{D}$ (*e.g.* random bitstrings in the example above). Intuitively, $\mathcal{D}$ represents the union of the sub-distributions of third round message for each Σ-protocol — enough of each *kind* of element that the simulators for each Σ-protocol can assemble a well-formed third round message from any element of $\mathcal{D}$. Any third round message for one of the Σ-protocols can be mapped into $\mathcal{D}$ by appending randomly sampled elements from the right sub-distributions to the message; inverting the mapping involves deterministically selecting the appropriate bits and dropping the rest.

Our cross-stacking compiler works as follows: the prover begins as in the self-stacking compiler, executing the first round message function of the active clause and computing a commitment using a partially-binding commitment scheme. After receiving the challenge, the prover honestly computes a third round message for the active clause. Next, the prover maps this message to some element d in the shared distribution $\mathcal{D}$. Finally, the prover extracts a third round message for each inactive clause from d, and simulates a transcript from this extracted message. The third round message then contains first round messages for each transcript, equivocating randomness, and d. The verifier uses the invertible mapping to extract a third round message for each clause, and verify these transcripts. The communication complexity of the compiler protocol is determined by the size of d. In Section 8, we show that this compiler can be efficiently applied to stack many Σ-protocols with each other, including MPC-in-the-head protocols like KKW [KKW18] and Ligero [AHIV17].

In Section 9, we briefly explore extended the ideas above to produce zero-knowledge proofs for proofs of partial knowledge, *i.e.* statements where the prover wishes to prove that it has witnesses to at least k out of the ℓ clauses. We note that solving this problem requires additional structure not present in the purely disjunctive setting. The communication complexity introduced by the compiler we present has an additive overhead that is linear in ℓ, making it less efficient than the other compilers we present in this work. We

[9] While it might be possible to define a Σ-protocol that uses different techniques for different parts of the relation, this would require the creation of a new, purpose built protocol — something we hope to avoid in this work. Thus, the difference between self-stacking in this work is primarily conceptual, rather than technical.

belive improving on this result is interesting future work.

Paper Organization. The paper is organized as follows: we present required preliminaries Section 4 and the interface for partially-binding commitment schemes in Section 5. In Section 6 we cover the properties of Σ-protocols that our compiler requires and give examples of conforming Σ-protocols. We present our self-stacking compiler in Section 7and our cross-stacking compiler in Section 8.Finally, in Section 9, we give an overview of extending our work to proofs of partial knowledge and in Section 10 we discuss the concrete efficiency of instantiating our compilers.

4 Preliminaries

4.1 Notation

Throughout this paper we use λ to denote the computational security parameter and κ to denote the statistical security parameter. We denote by $x \xleftarrow{\$} \mathcal{D}$ the sampling of 'x' from the distribution '$\mathcal{D}$'. We use $[n]$ as a short hand for a list containing the first n natrual numbers in order: i.e. $[n] = 1, 2 \ldots, n$. We denote by $x \xleftarrow{\$}_s \mathcal{D}$ the process of sampling 'x' from the distribution '$\mathcal{D}$' using pseudorandom coins derived from a PRG applied to the seed 's', when the expression occurs multiple times we mean that the element is sampled using random coins from disjoint parts of the PRG output. We denote by H a collision-resistant hash function (CRH). We write group operations using multiplicative notation.

4.2 Σ-Protocols

In this section, we recall the definition of a Σ-protocol.

Definition 1 (Σ-Protocol)**.** *Let $\mathcal{R}$ be an NP relation. A Σ-Protocol Π for $\mathcal{R}$ is a 3 move protocol between a prover P and a verifier V consisting of a tuple of PPT algorithms $\Pi = (A, Z, \phi)$ with the following interfaces:*

- *$a \leftarrow A(x, w; r^p)$: On input the statement x, corresponding witness w, such that $\mathcal{R}(x, w) = 1$, and prover randomness r^p, output the first message a that P sends to V in the first round.*
- *$c \xleftarrow{\$} \{0,1\}^\kappa$: Sample a random challenge c that V sends to P in the second round.*
- *$z \leftarrow Z(x, w, c; r^p)$: On input the statement x, the witness w, the challenge c, and prover randomness r^p, output the message z that P sends to V in the third round.*
- *$b \leftarrow \phi(x, a, c, z)$: On input the statement x, prover's messages a, z and the challenge c, this algorithm run by V, outputs a bit $b \in \{0, 1\}$.*

A Σ-protocol has the following properties:

- ***Completeness:*** *A Σ-Protocol $\Pi = (A, Z, \phi)$ is said to be complete if for any x, w such that $\mathcal{R}(x, w) = 1$, and any prover randomness $r^p \xleftarrow{\$} \{0,1\}^\lambda$, it holds that,*

$$\Pr\left[\phi(x, a, c, z) = 1 \;\middle|\; a \leftarrow A(x, w; r^p); c \xleftarrow{\$} \{0,1\}^\kappa; z \leftarrow Z(x, w, c; r^p)\right] = 1$$

- ***Special Soundness.*** *A Σ-Protocol $\Pi = (A, Z, \phi)$ is said to have special soundness if there exists a PPT extractor $\mathcal{E}$, such that given any two transcripts (x, a, c, z) and (x, a, c', z'), where $c \neq c'$ and $\phi(x, a, c, z) = \phi(x, a, c', z') = 1$, it holds that*

$$\Pr\left[R(x, w) = 1 |\; w \leftarrow \mathcal{E}(1^\lambda, x, a, c, z, c', z')\right] = 1$$

– ***Special Honest Verifier Zero-Knowledge.*** *A Σ-Protocol $\Pi = (A, Z, \phi)$ is said to be special honest verifier zero-knowledge, if there exists a PPT simulator $\mathcal{S}$, such that for any x, w such that $\mathcal{R}(x, w) = 1$, it holds that*

$$\{(a, z) \mid c \xleftarrow{\$} \{0,1\}^\kappa; (a, z) \leftarrow \mathcal{S}(1^\lambda, x, c)\} \approx_c \{(a, z) \mid r^p \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x, w; r^p); c \xleftarrow{\$} \{0,1\}^\kappa; z \leftarrow Z(x, w, c; r^p)\}$$

4.3 Secure Multiparty Computation

For completeness, we recall the definitions of t-privacy and t-robustness from [IKOS07] that will be used in MPC-in-the-head protocols.

Definition 2 (t-Privacy [IKOS07])**.** *Let $1 \leq t < n$. We say that Π realizes f with computational t-privacy if there is a PPT simulator $\mathcal{S}$ such that for any inputs $x, w_1, \ldots, w_n$ and every set of corrupt players $\mathcal{I} \subseteq [n]$ such that $|\mathcal{I}| \leq t$, the joint view $\mathsf{View}_{\mathcal{I}}(x, w_1, \ldots, w_n)$ of the players in $\mathcal{I}$ and $\mathcal{S}(\mathcal{I}, x, \{w_i\}_{i\in\mathcal{I}}, f(x, w_1, \ldots, w_n))$ are (identically distributed, statistically close, computationally close).*

Definition 3 (Statistical t-Robustness [IKOS07])**.** *Let $1 \leq t < n$. We say that Π realizes f with statistical t-robustness if (1) it correct evaluates f in the presence of a semi-honest adversary (with an most negligible error) and (2) if for any computationally unbounded malicious adversary corrupting a set $\mathcal{I}$ of at most t players, and for any inputs $(x, w_1, \ldots, w_n)$, if there is no $(w'_1, \ldots, w'_n)$ such that $f(x, w_1, \ldots, w_n) = 1$, then the probability that some uncorrupted player outputs 1 in an execution of Π in which the inputs of the honest player are consistent with $(x, w_1, \ldots, w_n)$ is negligible in the security parameter.*

5 Partially-Binding Vector Commitments

In this section, we introduce *non-interactive partially-binding vector commitments.*[10] These commitments allow a committer to commit to a vector of ℓ elements such that exactly t positions are binding (*i.e.* cannot be opened to another value) and the remaining $\ell - t$ positions can be equivocated. The committer must decide the binding positions of the vector before committing and the binding positions are hidden.

Definition 4 (t-out-of-ℓ Binding Vector Commitment)**.** *A t-out-of-ℓ binding non-interactive vector commitment scheme with message space $\mathcal{M}$, is defined by a tuple of the PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv}, \mathsf{BindCom})$ *defined as follows:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ *On input the security parameter λ, the setup algorithm outputs public parameters* pp.
- $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$*: Takes public parameters* pp *and a t-subset of indices $B \in \binom{[\ell]}{t}$. Returns a commitment key* ck *and equivocation key* ek.
- $(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}; r)$*: Takes public parameter* pp*, equivocation key* ek*, ℓ-tuple $\mathbf{v}$ and randomness r. Returns a partially-binding commitment* com *as well as some auxiliary equivocation information* aux.
- $r \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', \mathsf{aux})$*: Takes public parameters* pp*, equivocation key* ek*, original commitment value $\mathbf{v}$ and updated commitment values $\mathbf{v}'$ with $\forall i \in B : \mathbf{v}_i = \mathbf{v}'_i$, and auxiliary equivocation information* aux*. Returns equivocation randomness r.*
- $\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r)$*: Takes public parameters* pp*, commitment key* ck*, ℓ-tuple $\mathbf{v}$ and randomness r and outputs a commitment* com*. Note that this algorithm does not use the equivocation key* ek*. This algorithm plays a similar role to that of* Open *in a typical commitment scheme.*

[10] As pointed out in [ABFV22], there was a subtle issue in our definition, in the previous version of this paper. In their paper, Avitabile et al. [ABFV22] proposed a slight modification to our previous definition to help resolve the issue. In this updated version, we propose a slightly different modification than theirs to our previous definition that also helps resolve the issue observed in [ABFV22].

The properties satisfied by the above algorithms are as follows:

(Perfect) Hiding: *The commitment key* ck *and commitment* com *(perfectly) hides the binding positions* B *and the equivocated values, even when opening the commitment. Formally, for all* $\mathbf{v}^{(1)}, \mathbf{v}^{(2)} \in \mathcal{M}^\ell$*,* $B^{(1)}, B^{(2)} \in \binom{[\ell]}{t}$ *and a 'valid equivocation' for both vectors* $\mathbf{v}' \in \mathcal{M}^\ell$ *i.e.* $\forall i \in B^{(1)} : \mathbf{v}_i^{(1)} = \mathbf{v}'_i$ *and* $\forall i \in B^{(2)} : \mathbf{v}_i^{(2)} = \mathbf{v}'_i$*) and* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, the two distributions are equal:*

$$\left[(\mathsf{ck}, \mathsf{com}, r') \;\middle|\; \begin{array}{c} (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B^{(1)}); r \xleftarrow{\$} \{0,1\}^\lambda; \\ (\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}^{(1)}; r); \\ r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}^{(1)}, \mathbf{v}', \mathsf{aux}) \end{array} \right]$$

$$\stackrel{p}{=}$$

$$\left[(\mathsf{ck}, \mathsf{com}, r') \;\middle|\; \begin{array}{c} (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B^{(2)}); r \xleftarrow{\$} \{0,1\}^\lambda; \\ (\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}^{(2)}; r); \\ r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}^{(2)}, \mathbf{v}', \mathsf{aux}) \end{array} \right]$$

The definition essentially states that any two sets of binding positions and originally vectors, which could 'explain' the provided opening of $\mathbf{v}'$*, are indistinguishable.*

(Computational) Partial Binding: *An adversary (that generates* ck *itself) cannot equivocate on more than* $\ell - t$ *positions, even across multiple different commitments. Define the function* $\Delta : \mathcal{M}^\ell \times \mathcal{M}^\ell \mapsto \mathbb{P}([\ell])$ *taking two vectors and returning the set of indexes on which the vectors differ:*

$$\Delta(\mathbf{v}, \mathbf{v}') = \{j \in [\ell] : v_j \neq v'_j\} \ .$$

Consider an adversary $\mathcal{A}$ *that outputs* ck *and a set* S *of pairs of openings* $S \subseteq \mathcal{M}^\ell \times \mathcal{M}^\ell \times \mathcal{R} \times \mathcal{R}$ *such that each pair of openings share the same commitment under* ck*, then the set of index on which the openings differ across all pairs has cardinality at most* $t - \ell$*, formally, we require that the following probability is negligible in* λ *for any PPT* $\mathcal{A}$*:*

$$\Pr\left[\begin{array}{c} \left| \bigcup_{(\mathbf{v}, \mathbf{v}', r, r') \in S} \Delta(\mathbf{v}, \mathbf{v}') \right| > \ell - t \ \wedge \\ \forall (\mathbf{v}, \mathbf{v}', r, r') \in S.\ \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r) = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}'; r') \end{array} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{ck}, S) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) \end{array} \right] .$$

Partial Equivocation: *Given a commitment to* $\mathbf{v}$ *under a commitment key* $\mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$*, it is possible to equivocate to any* $\mathbf{v}'$ *as long as* $\forall i \in B : v_i = v'_i$*. More formally, for all* $B \in \binom{[\ell]}{t}$*, and all* $\mathbf{v}, \mathbf{v}' \in \mathcal{M}^\ell$ *st.* $\forall i \in B : v_i = v'_i$ *then:*

$$\Pr\left[\mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}'; r') = \mathsf{com} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); r \xleftarrow{\$} \{0,1\}^\lambda; \\ (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B); \\ (\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}; r); \\ r' \leftarrow \mathsf{Equiv}(\mathsf{ek}, \mathbf{v}, \mathbf{v}', \mathsf{aux}) \end{array} \right] = 1$$

Throughout this work we will impose the efficiency requirement that the size of the commitment is independent of the size of the elements. We note that this is easy to achieve using a collision resistant hash function, when targeting computational binding.

5.1 Relation To Similar Notions

The definition of partially binding vector commitments is similar to other definitions found in the literature, let us therefore briefly describe the distionguishing features of the definition above from these prior/concurrent definitions.

Somewhere Statistically Binding Hash Functions. A SSB (Somewhere Statistically Binding) hash function is a collision resistant hash function over vectors, it differs from partially binding commitments on many points: 1) a SSB hash function provide a short opening proof for any index. 2) the digest does not hide the vector, in particular does not provide equivocation for $i \neq i^*$. 3) the notion only considers a single binding index i^*. 4) the value at statistically binding index v_{i^*} can be extracted from the digest (extraction), which is not required for partially binding commitments.

Somewhere Statistically Binding Commitments. In concurrent work Fauzi, Lipmaa and Pindado [FLPS21] defines a very similar notion of a *somewhere statistically binding* commitment scheme, which provide hiding commitment to vectors and is binding in a subset of the indexes, however their definition/-motivation differs in a few important ways: 1) we do not require extraction, as a result partially binding commitments do not imply oblivious transfer unlike SSB commitments, indeed we know of constructions of partially binding commitments from non-blackbox commitments in the random oracle model. 2) we also do not require statistical binding for our compiler: there is a trade-off between statistical/computational binding of the partially binding vector commitment and computational/statistical zero-knowledge respectively of the compiled protocol; the instansiations we provide in this paper choose perfect hiding. 3) for our applications, the party sampling the commitment key generator cannot be trusted. 4) for efficiency we require $|\mathsf{ck}|$ to be small.

5.2 Partially-Binding Vector Commitments from Discrete Log

We now present a simple and concretely efficient construction of t-out-of-ℓ partially-binding vector commitments from the discrete log assumption. The idea is to have the committer use a Pedersen commitment for each element in the vector. Recall that a Pedersen commitment to the message $m \in \mathbb{Z}_{|\mathbb{G}|}$ with public parameters $g, h \in \mathbb{G}$ is computed as $g^m h^r$ for a random value r. The binding property of Pedersen commitments relies on the committer not knowing the discrete log of g with respect to h. For our partially-binding vector commitment scheme, the commitment key is a set of public parameters for the Pedersen commitments, constructed such a way that the committer knows discrete logs for exactly $\ell - t$ parameters. This is done by having the committer pick $\ell - t$ of the parameters and computing the remaining t parameters by interpolating in the exponent. More formally, let use begin by fixing some notation. Let $\mathbb{Z}_{|\mathbb{G}|}$ be a prime field. In our construction, we implicitly treat indexes $i \in [0, |\mathbb{G}| - 1]$ as field elements, *i.e.* there is an implicit bijective map between $[0, |\mathbb{G}| - 1]$ and $\mathbb{Z}_{|\mathbb{G}|}$ (e.g. $i \mod |\mathbb{G}| \in \mathbb{Z}/(|\mathbb{G}|)$). Let $\mathcal{X} \subseteq \mathbb{Z}_{|\mathbb{G}|}$ and $j \in \mathcal{X}$, define $L_{(\mathcal{X},j)}(X) \coloneqq \prod_{m \in \mathcal{X}, m \neq j} \frac{X-m}{j-m} \in \mathbb{Z}_{|\mathbb{G}|}[X]$ i.e. the unique degree $|\mathcal{X}| - 1$ polynomial for which $\forall x \in \mathcal{X} \setminus \{j\} : L_{(\mathcal{X},j)}(x) = 0$ and $L_{(\mathcal{X},j)}(j) = 1$. The formal description of the commitment scheme can be found in Figure 2. While our construction does require a CRS, we note that the CRS is just two randomly selected group elements[11], which in practice can be generated by hashing a 'nothing-up-by-sleeve' constant to the curve by using a cryptographic hash function.

Theorem 1. *Under the discrete log assumption (Definition 5), for any (t, ℓ) with $t < \ell$: the scheme shown in Figure 2 is a family of (perfectly hiding, computationally binding) t-of-ℓ partially binding commitment schemes.*

The security reduction is straightforward and tight: for each position i in which the adversary $\mathcal{A}$ manages to equivocate we can extract the discrete log of g_i (as for regular Pedersen commitments), if we extract the discrete log in $\ell - t + 1$ positions, we have sufficient points on the degree $\ell - t$ polynomial to recover $f_{[\ell] \cup \{0\}}(X)$ explicitly and simply evaluate it at 0 to recover the discrete log of g_0 from pp. We present a formal description of this reduction to the discrete log assumption below.

[11] Like regular Pedersen commitments

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$

1 : $\mathbb{G} \leftarrow \mathsf{GenGroup}(1^\lambda); g_0, h \xleftarrow{\$} \mathbb{G}$
2 : **return** $(\mathbb{G}, g_0, h)$

$(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v})$:

1 : $r \xleftarrow{\$} \mathbb{Z}^\ell_{|\mathbb{G}|}$
2 : $\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}, r)$
3 : **return** (com, r)

$(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$

1 : Let $E = [\ell] \setminus B$ (set of equivocal indexes)
2 : Generate trapdoors for $\ell - t$ indexes: **for** $i \in E : y_i \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}, g_i \leftarrow h^{y_i}$
3 : Interpolate the first $_{[\ell-t]}$ elements: **for** $j \in [\ell - t] : g_j \leftarrow \prod_{i \in E \cup \{0\}} {g_i}^{L_{(E\cup\{0\},i)}(j)}$
4 : $\mathsf{ck} = (g_1, \ldots, g_{\ell-t})$
5 : $\mathsf{ek} = (g_1, \ldots, g_{\ell-t}, \{y_i\}_{i\in E}, E, B)$
6 : **return** $(\mathsf{ck}, \mathsf{ek})$

$r \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', \mathsf{aux})$:

1 : Let $E = [\ell] \setminus B$ (set of equivocal indexes)
2 : Parse $\mathsf{aux} = (r_1, \ldots, r_\ell) \in \mathbb{Z}^\ell_{|\mathbb{G}|}$
3 : Interpolate the remaining elements: **for** $j \in [\ell - t, \ell] : g_j \leftarrow \prod_{i \in [\ell-t]\cup\{0\}} g_i^{L_{([\ell-t]\cup\{0\},i)}(j)}$
4 : **for** $j \in B : r'_j \leftarrow r_j$
5 : **for** $j \in E : r'_j \leftarrow r_j - y_j \cdot ({\mathbf{v}'}_j - \mathbf{v}_j)$
6 : **return** r'

$\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}, r)$:

1 : Interpolate the remaining elements: **for** $j \in [\ell - t, \ell] : g_j \leftarrow \prod_{i \in [\ell-t]\cup\{0\}} g_i^{L_{([\ell-t]\cup\{0\},i)}(j)} \in \mathbb{G}$
2 : Commit individually: **for** $j \in [\ell] : \mathsf{com}_j \leftarrow h^{r_j} \cdot g_j^{\mathbf{v}_j} \in \mathbb{G}$
3 : **return** $(\mathsf{com}_1, \ldots, \mathsf{com}_\ell)$

Figure 2: t-of-ℓ binding commitment from discrete log in the CRS model.

Remark 1. *To commit to longer strings a collision resistant hash* $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_{|\mathbb{G}|}$ *is used to compress each coordinate before committing using* BindCom/EquivCom *as a black-box: by committing to* $\mathbf{v}' = (\mathsf{H}(v_1), \ldots, \mathsf{H}(v_\ell))$ *instead. Note that the discrete log assumption (Definition 5), used above, also implies the existance of collision resistant hash functions.*

Definition 5 (Discrete Log Assumption)**.** *There exists a PPT algorithm* $\mathsf{GenGroup}(1^\lambda)$ *which returns a description of a prime-order cyclic group* $\mathbb{G}$ *(written multiplicatively) which admits efficient sampling, st. for all PPT algorithms* $\mathcal{A}$*:* $\Pr\left[\mathcal{A}(1^\lambda, \mathbb{G}, g, h) = y \mid \mathbb{G} \leftarrow \mathsf{GenGroup}(1^\lambda); h \xleftarrow{\$} \mathbb{G}; y \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}; g \leftarrow h^y\right] = \mathsf{negl}(\lambda)$*. For some negligible function* $\mathsf{negl}(\lambda)$*.*

Theorem 1. Completeness of partial equivocation for the scheme in Figure 2 is easily seen (follows from equivocation of Pedersen commitments), so we focus on computational binding and perfect hiding.

Computational Binding Let $\mathcal{A}_k$ be a PPT algorithm winning the binding game with probability ϵ i.e.

$$\epsilon = \Pr\left[\begin{array}{c} \nexists S \subset [\ell], |S| \geq t, \text{ s.t. } i \in S, v_{1,i} = \ldots = v_{k,i} \ \wedge \\ \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}_1; r_1) = \ldots = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}_k; r_k) \end{array} \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); \\ (\mathsf{ck}, \mathbf{v}_1, \ldots, \mathbf{v}_k, r_1, \ldots, r_k) \leftarrow \mathcal{A}_k(1^\lambda, \mathsf{pp}) \end{array}\right]$$

Then the PPT algorithm $\mathcal{A}'$ shown in Figure 3 wins the discrete log game (computing y_0 st. $g_0 = h^{y_0}$) with probability $\geq \epsilon$. To see this observe that, when $\mathcal{A}_k$ wins the binding game: it follows that there exists a set S such that its complement $\overline{S}$ has size $|\overline{S}| \geq \ell - t + 1$ and since $\forall \alpha, \beta \in [k]$: $(\mathsf{com}_1, \ldots, \mathsf{com}_\ell) = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}^{(\alpha)}) = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}^{(\beta)}) \in \mathbb{G}^\ell$, we can extract $y_i \in \mathbb{Z}_{|\mathbb{G}|}$ st. $g_i = h^{y_i}$ whenever $\mathbf{v}_i^{(\alpha)} \neq \mathbf{v}_i^{(\beta)}$ by observing:

$$\begin{aligned} g_i^{\mathbf{v}_i^{(\alpha)}} h^{\mathbf{r}_i^{(\alpha)}} &= \mathsf{com}_i = g_i^{\mathbf{v}_i^{(\beta)}} h^{\mathbf{r}_i^{(\beta)}} \\ g_i^{\mathbf{v}_i^{(\beta)} - \mathbf{v}_i^{(\alpha)}} &= h^{\mathbf{r}_i^{(\alpha)} - \mathbf{r}_i^{(\beta)}} \\ g_i = h^{y_i} &= h^{(\mathbf{r}_i^{(\alpha)} - \mathbf{r}_i^{(\beta)})/(\mathbf{v}_i^{(\beta)} - \mathbf{v}_i^{(\alpha)})} \end{aligned}$$

Consider $\mathcal{X} \subseteq_{\ell-t+1} \overline{S}$ defined as in $\mathcal{A}'$, let $f_{\mathcal{X}}(X) \coloneqq \sum_{i \in \mathcal{X}} y_i \cdot L_{(\mathcal{X},i)}(X) \in \mathbb{Z}_{|\mathbb{G}|}[X]$. Consider $f_{[\ell-t]\cup\{0\}}(X) \coloneqq \sum_{i \in [\ell-t]\cup\{0\}} y_i \cdot L_{([\ell-t]\cup\{0\},i)}(X)$ defined by the unique $y_0, y_1, \ldots, y_{\ell-t} \in \mathbb{Z}_{|\mathbb{G}|}$ with $g_0 = h^{y_0}, \ldots, g_1 = h^{y_1}, \ldots, g_{\ell-t} = h^{y_{\ell-t}}$ where $\mathsf{ck} = (g_1, \ldots, g_{\ell-t})$. Observe that $\forall j \in \mathcal{X} : f_{\mathcal{X}}(j) = f_{[\ell]\cup\{0\}}(j)$ hence $f_{\mathcal{X}} = f_{[\ell-t]\cup\{0\}}$ since both are degree $\ell - t < |\mathcal{X}|$ polynomials. Therefore the algorithm recovers $f_{\mathcal{X}}(0) = f_{[\ell]\cup\{0\}}(0) = \sum_{i \in \mathcal{X}} y_i \cdot L_{(\mathcal{X},i)}(0) = y_0$, with $g_0 = h^{y_0}$, by definition of $f_{[\ell]\cup\{0\}}$.

$y_0 \leftarrow \mathcal{A}'^{\mathcal{A}_k}(1^\lambda, \mathbb{G}, g_0, h)$: computes the discrete log of g_0 in h given oracle access to $\mathcal{A}_k$.

1 : Let $\mathsf{pp} = (\mathbb{G}, g_0, h)$
2 : $(\mathsf{ck}, \mathbf{v}^{(1)}, \ldots \mathbf{v}^{(k)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k)}) \leftarrow \mathcal{A}_k(1^\lambda, \mathsf{pp})$
3 : $\overline{S} = \{i \mid \exists\ (\alpha, \beta) : \mathbf{v}_i^{(\alpha)} \neq \mathbf{v}_i^{(\beta)}\} \subseteq [\ell]$, **if** $|\overline{S}| \leq \ell - t$: **return** $\perp$
4 : **for** $i \in \overline{S}$ compute the discrete log in h: $y_i \leftarrow (\mathbf{r}_i^{(\alpha)} - \mathbf{r}_i^{(\beta)})/(\mathbf{v}_i^{(\beta)} - \mathbf{v}_i^{(\alpha)})$
5 : Pick $\mathcal{X} \subseteq_{\ell-t+1} \overline{S}$, compute $y_0 \leftarrow \sum_{i \in \mathcal{X}} y_i \cdot L_{(\mathcal{X},i)}(0)$
6 : **return** y_0

Figure 3: Reduction for partially binding commitment scheme to discrete log.

Note that the security reduction is tight.

Perfect Hiding Recall that we denote the set of binding indexes as B, and its complement (the set of indexes that support equivocation) as E. Observe that for any E the distribution of $\mathsf{ck} = (g_1, \ldots, g_{[\ell]-t})$ is uniform in $\mathbb{G}^{\ell-t}$: since the distribution of $\{g_j\}_{j \in E}$ is uniform and $\{g_j\}_{j \in [\ell-t]}$ is computed as a bijection of $\{g_j\}_{j \in E}$. Hence the distribution of ck is independent of E (and B), and the binding indexes are perfectly hidden. The perfect hiding of the commitment $(\mathsf{com}_1, \ldots, \mathsf{com}_\ell)$ follows directly from perfect hiding of Pedersen commitments: each com_i is sampled i.i.d. uniform from $\mathbb{G}$.

□

5.3 Generic Construction of 1-of-2^q Partially-Binding Vector Commitment.

From a 1-of-2 partial-binding vector commitment scheme, it is easy to obtain a 1-of-2^q binding scheme in which the communication complexity grows linearly in q (i.e. logarithmically in the dimension of the vector), this is achieved by computing a tree of commitments in which the leafs are the entries of the vector being committed to and each internal node is formed by committing to its children: other commitments. There is one commitment key per level and the final commitment is the root of the tree. This means that the binding indexes in the commitment keys encode a path though the tree, leading to a single binding leaf, where as every other path though the tree can be equivocated at some layer. To formalize this, we describe the case of a tree with just two layers, as described below, then apply the transformation iteratively:

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$

1 : $\mathsf{pp}_A \leftarrow \mathsf{PBComm}_A.\mathsf{Setup}(1^\lambda)$
2 : $\mathsf{pp}_B \leftarrow \mathsf{PBComm}_B.\mathsf{Setup}(1^\lambda)$
3 : **return** $(\mathsf{pp}_A, \mathsf{pp}_B)$

$(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp} = (\mathsf{pp}_A, \mathsf{pp}_B), B = \{i\})$

1 : Compute $i_A = i \mod \ell_A, i_B = \lfloor i/\ell_A \rfloor$
2 : $(\mathsf{ck}_A, \mathsf{ek}_A) \leftarrow \mathsf{PBComm}_A.\mathsf{Gen}(\mathsf{pp}_A, \{i_A\})$
3 : $(\mathsf{ck}_B, \mathsf{ek}_B) \leftarrow \mathsf{PBComm}_B.\mathsf{Gen}(\mathsf{pp}_A, \{i_B\})$
4 : **return** $(\mathsf{ck}_A, \mathsf{ck}_B), (\mathsf{ek}_A, \mathsf{ek}_B, i)$

$(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp} = (\mathsf{pp}_A, \mathsf{pp}_B), \mathsf{ek} = (\mathsf{ek}_A, \mathsf{ek}_B, i), \mathbf{v})$:

1 : Compute $i_A = i \mod \ell_A, i_B = \lfloor i/\ell_A \rfloor$
// Form a length ℓ_A vector v_A, which is zero everywhere except at i_A where it is v_i
2 : $\mathbf{v}^{(\mathbf{A})} \leftarrow \mathbf{0}; v^{(A)}_{i_A} \leftarrow v_i$
3 : $(\mathsf{com}_A, \mathsf{aux}_A) \leftarrow \mathsf{PBComm}_A.\mathsf{EquivCom}(\mathsf{pp}_A, \mathsf{ek}_A, \mathbf{v}^{(\mathbf{A})})$
// Form a length ℓ_B vector v_B, which is zero everywhere except at i_B where it is com_A
4 : $\mathbf{v}^{(\mathbf{B})} \leftarrow \mathbf{0}; v^{(B)}_{i_B} \leftarrow \mathsf{com}_A$
5 : $(\mathsf{com}_B, \mathsf{aux}_B) \leftarrow \mathsf{PBComm}_B.\mathsf{EquivCom}(\mathsf{pp}_B, \mathsf{ek}_B, \mathbf{v}^{(\mathbf{B})})$
// Return the root/outer commitment
6 : **return** $(\mathsf{com}_B, (\mathsf{aux}_A, \mathsf{aux}_B))$

$r \leftarrow \mathsf{Equiv}(\mathsf{pp} = (\mathsf{pp}_A, \mathsf{pp}_B), \mathsf{ek} = (\mathsf{ek}_A, \mathsf{ek}_B, i), \mathbf{v}, \mathbf{v}', \mathsf{aux} = (\mathsf{aux}_A, \mathsf{aux}_B))$:

1 : Compute $i_A = i \mod \ell_A, i_B = \lfloor i/\ell_A \rfloor$
// Equivocate the inner commitment
2 : $\mathbf{v}^{(\mathbf{A})} \leftarrow \mathbf{0}; v^{(A)}_{i_A} \leftarrow v_i$
3 : $\mathbf{v}^{(\mathbf{A})\prime} \leftarrow (v'_{i_B \ell_B + 1}, \ldots, v'_{(i_B+1)\ell_B})$ // Note $v^{(A)\prime}_{i_A} = v^{(A)}_{i_A}$
4 : $r_A \leftarrow \mathsf{PBComm}_A.\mathsf{Equiv}(\mathsf{pp}_A, \mathsf{ek}_A, \mathbf{v}^{(\mathbf{A})}, \mathbf{v}^{(\mathbf{A})\prime})$
// Recompute $\mathbf{v}^{(\mathbf{B})}$
5 : $\mathbf{v}^{(\mathbf{B})} \leftarrow \mathbf{0}; v^{(B)}_{i_B} \leftarrow \mathsf{PBComm}_A.\mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}_A, \mathbf{v}^{(\mathbf{A})}, r_A)$
// Commit to every chunk using the same randomness r_A to obtain $\mathbf{v}^{(\mathbf{B})\prime}$
6 : $\mathbf{v}^{(\mathbf{B})\prime} \leftarrow \mathbf{0}$; **for** $j \in 1, \ldots, \ell_B$:
7 : $\hat{v'}_j \leftarrow (v'_{j\ell_B+1}, \ldots, v'_{(j+1)\ell_B})$ // Next "chunk" of $\mathbf{v}'$
8 : $v^{(B)\prime}_j \leftarrow \mathsf{PBComm}_A.\mathsf{BindCom}(\mathsf{pp}_A, \mathsf{ck}_A, \hat{v'}_j, r_A)$
// Note $v^{(B)\prime}_{i_B} = v^{(B)}_{i_B}$. Equivocate the outer commitment
9 : $r_B \leftarrow \mathsf{PBComm}_B.\mathsf{Equiv}(\mathsf{pp}_B, \mathsf{ek}_B, \mathbf{v}^{(\mathbf{B})}, \mathbf{v}^{(\mathbf{B})\prime})$
10 : **return** (r_A, r_B)

$\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp} = (\mathsf{pp}_A, \mathsf{pp}_B), \mathsf{ck} = (\mathsf{ck}_A, \mathsf{ck}_B), \mathbf{v}, r = (r_A, r_B))$:

// Commit to every chunk using the same randomness r_A to obtain $\mathbf{v}^{(\mathbf{B})}$
1 : $\mathbf{v}^{(\mathbf{B})} \leftarrow \mathbf{0}$; **for** $j \in 1, \ldots, \ell_B$:
2 : $\hat{v}_j \leftarrow (v_{j\ell_B+1}, \ldots, v_{(j+1)\ell_B})$ // Next "chunk" of $\mathbf{v}$
3 : $v^{(B)}_j \leftarrow \mathsf{PBComm}_A.\mathsf{BindCom}(\mathsf{pp}_A, \mathsf{ck}_A, \hat{v}_j, r_A)$
// Commit to the vector of commitments $\mathbf{v}^{(\mathbf{B})}$ to obtain the final commitment
4 : **return** $\mathsf{PBComm}_B.\mathsf{BindCom}(\mathsf{pp}_B, \mathsf{ck}_B, \mathbf{v}^{(\mathbf{B})}, r_B)$

Figure 4: Generic construction of 1-of-$\ell_A\ell_B$ binding commitment from a 1-of-ℓ_A binding commitment scheme and a 1-of-ℓ_B binding commitment scheme.

Theorem 2 (1-of-$(\ell_A\ell_B)$ partial-binding from 1-of-ℓ_A and 1-of-ℓ_B partial-binding.)**.** *Given compressing* 1*-of-*ℓ_A *and* 1*-of-*ℓ_B *partial-binding vector commitment schemes* PBComm_A, PBComm_B *with communication complexity* $\mathtt{CC}(\mathsf{PBComm}_A)$ *and* $\mathtt{CC}(\mathsf{PBComm}_B)$ *respectively, there exists a 1-of-*$(\ell_A \cdot \ell_B)$ *partial-binding vector commitment with communication complexity* $\mathtt{CC}(\mathsf{PBComm}_A) + \mathtt{CC}(\mathsf{PBComm}_B)$ *making black-box use of the underlying* PBComm_A *and* PBComm_B*.*

Proof. The construction works by forming partially binding commitments to partially binding commitments as follows: 1) split the vector $\mathbf{v}$ into ℓ_B chunks $\hat{v}_1, \ldots, \hat{v}_{\ell_B}$ of size ℓ_A each, 2) commit to each chunk individually using PBComm_A with *the same commitment key* ck_A, obtain commitment $\mathbf{v}^{(\mathbf{B})} = (\mathsf{com}_1, \ldots, \mathsf{com}_{\ell_B})$, 3) commit to the commitments $\mathbf{v}^{(\mathbf{B})}$ using PBComm_B and corresponding commitment key ck_B. This scheme is formally described in Figure 4. Binding and hiding follows easily from binding and hiding respectively of PBComm_A and PBComm_B. □

By applying this transformation iteratively q times to a 1-of-2 binding scheme, we obtain a 1-of-2^q binding scheme with communication linear in q.

Corollary 1. *There exists a (concretely efficient)* 1*-of-*2^q *binding commitment scheme with* $O(\lambda \cdot q)$*-communication (for committing and opening) from the discrete log assumption.*

Proof. Apply the transformation from Figure 4 q times iteratively to the scheme from Theorem 1 with $\ell = 2$ and $t = 1$. i.e. let the original scheme from Theorem 1 be PBComm_1, compose PBComm_1 with itself to obtain a new 1-of-2^2 binding scheme PBComm_2, then compose PBComm_2 with PBComm_1 to obtain a 1-of-2^3 binding scheme PBComm_3, then compose PBComm_3 with PBComm_1 to obtain a 1-of-2^4 binding scheme PBComm_4, etc. At every step the communication grows by $\mathtt{CC}(\mathsf{PBComm}_1)$, hence the communication of PBComm_q is $q \cdot \mathtt{CC}(\mathsf{PBComm}_1)$. □

6 Stackable Σ-Protocols

In this section, we present the properties of Σ-protocols that our stacking framework requires and show that many Σ-protocols satisfy these properties.

6.1 Properties of Stackable Σ-Protocols.

We start by formalizing the definition of a "stackable" Σ-protocol. As discussed in Section 3, a Σ-protocol is stackable (meaning, it can be used by our stacking framework), if it satisfies two main properties: (1) simulation with respect to a specific third round message, and (2) recyclable third round messages.

6.1.1 Cheat Property: "Extended" Honest Verifier Zero-Knowledge.

We view "simulation with respect to a specific third round message" as a natural strengthening of the typical special honest verifier zero-knowledge property of Σ-protocols. At a high level, this property requires that it is possible to design a simulator for the Σ-protocol by first sampling a random third round message from the space of admissible third round messages, and then constructing the unique appropriate first round message. We refer to such a simulator as an *extended simulator*. A similar notion is considered by Abe *et. al* [AOS02] in their definition of `type-T` signature schemes: a `type-T` signature scheme is essentially the Fiat-Shamir [FS87] heuristic applied to an EHVZK Σ-protocol.

Definition 6 (EHVZK Σ-Protocol)**.** *Let* $\Pi = (A, Z, \phi)$ *be a* Σ*-protocol for the NP relation* $\mathcal{R}$*, with a well-behaved simulator. We say that* Π *is "extended honest-verifier zero-knowledge (EHVZK)" if there exists a polynomial time computable* <u>*deterministic*</u> "extended simulator" $\mathcal{S}^{\text{EHVZK}}$ such that for any $(x, w) \in \mathcal{R}$ and $c \in \{0,1\}^\kappa$, there exists an efficiently samplable distribution $\mathcal{D}^{(z)}_{x,c}$ such that:

$$\left\{(a, c, z) \mid r^p \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p)\right\} \approx \left\{(a, c, z) \mid z \xleftarrow{\$} \mathcal{D}^{(z)}_{x,c}; a \leftarrow \mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z)\right\}$$

The natural variants (perfect/statistical/computational) of EHVZK are defined depending on which class of distinguishers for which $\approx$ is defined.

At first glace, the EHVZK definition can appear contrived, however in practice this is often how simulators for Σ-protocols are constructed: picking a third message z for a given challenge c, then finding the first round message a which 'matches' without relying on the random coins needed to sample z. For instance, every 'commit-and-open' Σ-protocol is EHVZK; this notably includes every protocol derived via IKOS [IKOS07]. Despite this, we note that there exist Σ-protocols which are not EHVZK in their natural form: consider a contrived Σ-protocol where z contains the output of a one-way function evaluated on a; an extended simulator for such a protocol would need to invert the one-way function. While clearly such protocols exist, to our knowledge, none are of practical importance. Nevertheless, we observe that it is possible to trivially compile any Σ-protocol into one for the same relation which is EHVZK.

Observation 1 (All Σ-protocols can be made EHVZK.)**.** *Any Σ-protocol $\Pi = (A, Z, \phi)$ can be transformed into an EHVZK Σ-protocol $\Pi' = (A', Z', \phi')$ for the same relation. We present one such transformation below:*

$A'(x, w; r^{\mathsf{P}}) \mapsto a'$

1 : *Run* $a \leftarrow A(x, w, r^{\mathsf{P}})$

2 : **return** a

$Z'(x, w, c'; r^{\mathsf{P}}) \mapsto z'$

1 : *Run* $z \leftarrow Z(x, w, c'; r^{\mathsf{P}})$

2 : *Run* $a \leftarrow A(x, w; r^{\mathsf{P}})$

3 : **return** (a, z)

$\phi'(x, a', c', z') \mapsto \{0, 1\}$

1 : *Parse* $z' = (a, z)$

2 : **return** $(a \stackrel{?}{=} a') \wedge \phi(x, a, c', z)$

$\mathcal{S}^{\text{EHVZK}\prime}(x, c', z') \mapsto a'$

1 : *Parse* $z' = (a, z)$

2 : **return** a

The transformation above simply uses the prover's randomness to re-generate the first round message a and appends it to the third round message, so the resulting third round message is (a, z). The verifier additionally checks that the a's contained in the first round message and third round message match. Defining the extended simulator for this transformed protocol is trivial: because the third round message contains a copy of the first round message, the extended simulator need only parse it out and return it. In this case, $\mathcal{D}^{(z)}_{x,c}$ is simply the output distribution of the Special Honest-Verifier Zero-Knowledge simulator of Π. By construction, it is clear that the protocol above is EHVZK for any Σ-protocol Π.

The challenge dependence on the distribution $\mathcal{D}^{(z)}_{x,c}$ might at first glance seem inherent, as it is possible for Σ-protocols to have very different third round message distributions depending on the challenge. Consider, for example, the Blum's three round graph Hamiltonicity Σ-protocol [Blu87]. The third round message is either a Hamiltonian path or a graph isomorphism, depending on the challenge, which can be represented with very different distributions. However, it would be convenient, both notationally and conceptually, to remove this dependence from the definition of EHVZK. We note that another simple transformation can be applied to any EHVZK Σ-protocol such that it satisfies a challenge-independent version of Definition 6. This observation is similar to that of Cramer *et. al* [CDS94] in relation to SHVZK from HVZK.

Definition 7 (Challenge-independent EHVZK Σ-Protocol)**.** *Let $\Pi = (A, Z, \phi)$ be a Σ-protocol for the NP relation $\mathcal{R}$. We say that Π is "challenge-independent extended honest-verifier zero-knowledge " if there exists a polynomial time computable deterministic "*challenge-independent extended simulator" $\mathcal{S}^{\text{CIEHVZK}}$ such that for any $(x, w) \in \mathcal{R}$ there exists an efficiently samplable distribution $\mathcal{D}^{(z)}_x$ such that:

$$\left\{(a, c, z) \mid r^p \stackrel{\$}{\leftarrow} \{0,1\}^\lambda; a \leftarrow A(x, w; r^p); z \leftarrow Z(x, w, c; r^p)\right\} \approx \left\{(a, c, z) \mid z \stackrel{\$}{\leftarrow} \mathcal{D}^{(z)}_x; a \leftarrow \mathcal{S}^{\text{CIEHVZK}}(1^\lambda, x, c, z)\right\}$$

Observation 2 (EHVZK Σ-protocol to challenge-independent EHVZK)**.** *We note that any EHVZK Σ-protocol can be transformed to be challenge-independent EHVZK. Let $\Pi = (A, Z, \phi)$ be an EHVZK Σ-protocol*

with a 'challenge-dependent' distribution $\mathcal{D}^{(z)}_{x,c}$ over last-round messages and define $\Pi' = (A', Z', \phi')$ as shown below:

$A'(x, w; r^{\mathsf{P}}) \mapsto a'$	$Z'(x, w, c'; r^{\mathsf{P}}) \mapsto z'$
Sample $\Delta \xleftarrow{\$} \{0,1\}^{\kappa}$	*Define* $c = c' \oplus \Delta$
Run $a \leftarrow A(x, w; r^{\mathsf{P}})$	*Run* $z \leftarrow Z(x, w, c; r^{\mathsf{P}})$
return (a, Δ)	**return** (z, c)

$\phi'(x, a', c', z') \mapsto \{0,1\}$	$\mathcal{S}^{\text{CIEHVZK}'}(x, c', z') \mapsto a'$
Parse $a' = (a, \Delta), z' = (z, \_)$	*Parse* $z' = (z, c)$
Define $c = c' \oplus \Delta$	*Define* $\Delta = c \oplus c'$
return $\phi(x, w, c, z)$	*Run* $a \leftarrow \mathcal{S}^{\text{EHVZK}}(x, c, z)$
	return (a, Δ)

In this transformation, we append a random string Δ to the first round message. The third round message algorithm then xor's Δ with the challenge provided by the verifier before computing the third round message z. Additionally, it appends the resulting challenge c to the third round message. The verifier recomputes c and verifies the transcript using c. The resulting protocol Π' satisfies Definition 7, i.e. the family $\mathcal{D}^{(z)'}_{x,c}$ has the same distribution across all c. To sample from the distribution $\mathcal{D}^{(z)}_{x}$, simply sample $c \xleftarrow{\$} \{0,1\}^{\kappa}$ randomly, then sample from $\mathcal{D}^{(z)}_{x,c}$ i.e.

$$\mathcal{D}^{(z)}_{x} \coloneqq \left\{ (z, c) \mid c \xleftarrow{\$} \{0,1\}^{\kappa}; z \xleftarrow{\$} \mathcal{D}^{(z)}_{x,c} \right\}$$

The challenge-independent extended simulator $\mathcal{S}^{\text{CIEHVZK}'}$ of Π' picks Δ such that the difference between c' and Δ is c and runs the extended simulator of Π on z with challenge $c = c' \oplus \Delta$.

It is straightforward to move from one definition to the other (by applying the compiler in Observation 2), however many existing Σ-protocol are naturally EHVZK for Definition 6 and therefore it is more convenient to use this more relaxed definition when showing that particular Σ-protocols are EHVZK. As such, for the remainder of the main body of this work, we will use Definition 6.

6.1.2 Re-use Property: Recyclable Third Round Messages.

The next property that our stacking compilers require is that the distribution of third round messages does not significantly rely on the statement. In more detail, given a fixed challenge, the distribution of possible third round messages for any pair of statements in the language are indistinguishable from each other. We formalize this property by using $\mathcal{D}^{(z)}_{c}$ to denote a single distribution with respect to a fixed challenge c. We say that a Σ-protocol has recyclable third round messages, if for any statement x in the language the distribution of all possible third round messages corresponding to challenge c is indistinguishable from $\mathcal{D}^{(z)}_{c}$. We now formally define this property:

Definition 8 (Σ-Protocol with Recyclable Third Messages)**.** *Let $\mathcal{R}$ be an NP relation and $\Pi = (A, Z, \phi)$ be a Σ-protocol for $\mathcal{R}$, with a well-behaved simulator. We say that Π has* recyclable third messages *if for each $c \in \{0,1\}^{\kappa}$, there exists an efficiently sampleable distribution $\mathcal{D}^{(z)}_{c}$, such that for all instance-witness pairs (x, w) st. $\mathcal{R}(x, w) = 1$, it holds that*

$$\mathcal{D}^{(z)}_{c} \approx \left\{ z \mid r^{p} \xleftarrow{\$} \{0,1\}^{\lambda}; a \leftarrow A(x, w; r^{p}); z \leftarrow Z(x, w, c; r^{p}) \right\}.$$

This property is fundamental to stacking, as it means that the contents of the third round message do not 'leak information' about the statement used to generate the message. This means that the message can be

safely re-used to generate transcripts for the non-active clauses and an adversary cannot detect which clause is active.[12] Although this property might seem strange, we will later show that many natural Σ-protocols have this property.

6.1.3 Stackability

With our two-properties formally defined, we are now ready to present the definition of stackable Σ-protocols:

Definition 9 (Stackable Σ-Protocol)**.** *We say that a Σ-protocol $\Sigma = (A, Z, \phi)$ is* stackable*, if it is EHVZK (see Definition 6) and has recyclable third messages (see Definition 8).*

We now note a useful property of stackable Σ-protocols that follow directly from Definition 9:

Remark 2. *Let $\Sigma = (A, Z, \phi)$ be a* stackable *Σ-protocol for the NP relation $\mathcal{R}$, with a well-behaved simulator. Then for each $c \in \{0,1\}^\lambda$ and any instance-witness pair (x, w) with $\mathcal{R}(x, w) = 1$, an honestly computed transcript is computationally indistinguishable from a transcript generated by sampling a random third round message from $\mathcal{D}_c^{(z)}$ and then simulating the remaining transcript using the extended simulator. More formally,*

$$\left\{(a,z) \mid r^p \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow A(x,w;r^p); z \leftarrow Z(x,w,c;r^p)\right\} \approx \left\{(a,z) \mid z \xleftarrow{\$} \mathcal{D}_c^{(z)}; a \leftarrow \mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z)\right\}$$

Looking ahead, these observations will be critical in proving security of our compilers in Sections 7 and 8.

6.2 Classical Examples of Stackable Σ-Protocols

In this section, we show some examples of classical Σ-protocols which are stackable. Rather than considering multiple classical Σ-protocols like Schnorr and Guillou-Quisquater separately, we consider the generalization of these protocols as explored in [CD98]. Once we show that this generalization is stackable, it is simple to see that specific instantiations are also stackable.

Lemma 1 (Σ-protocol for ψ-preimages [CD98] is stackable)**.** *Let $\mathfrak{G}_1^*$ and $\mathfrak{G}_2^*$ be groups with group operations $*_1, *_2$ respectively (multiplicative notation) and let $\psi : \mathfrak{G}_1^* \to \mathfrak{G}_2^*$ be a one-way group-homomorphism. Recall the simple Σ-protocol (Π_ψ) of Cramer and Damgård [CD98] for the relation of preimages $\mathcal{R}_\psi(x, w) \coloneqq x \stackrel{?}{=} \psi(w)$, where $x \in \mathfrak{G}_2^*, w \in \mathfrak{G}_1^*$. The protocol is a generalization of Schnorr [Sch90] and works as follows:*

- *$A(x, w; r^p)$, the prover samples $r \xleftarrow{\$} \mathfrak{G}_1^*$ and sends the image $a = \psi(r) \in \mathfrak{G}_2^*$ to the verifier.*
- *$Z(x, w, c; r^p)$, the prover intreprets c as an integer from a subset $C \subseteq \mathbb{Z}$ and replies with $z = w^c *_1 r$*
- *$\phi(x, a, c, z)$, the verifier checks $\psi(z) = x^c *_2 a$.*

Completeness follows since ψ is a homomorphism: $\psi(z) = \psi(w^c *_1 r) = \psi(w)^c *_2 \psi(r) = x^c *_2 a$. The knowledge soundness error is $1/|C|$ (see [CD98] for more details). For any homomorphism ψ, Π_ψ is stackable:

Proof. To see that Π_ψ is stackable, define an extended simulator and check for recyclable third messages:

1. Π_ψ is EHVZK: Let $\mathcal{D}_{x,c}^{(z)} \coloneqq \{z \mid z \xleftarrow{\$} \mathfrak{G}_1^*\}$, let $\mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z) \coloneqq \psi(z) *_2 x^{-c}$
2. Π_ψ has recyclable third messages: Observe that $\forall x_1, x_2 : \mathcal{D}_{x_1,c}^{(z)} = \mathcal{D}_{x_2,c}^{(z)} = \mathcal{U}(\mathfrak{G}_1^*)$[13].

□

Remark 3. *The following variants of Π_ψ (with different choices of $\mathfrak{G}_1^*, \mathfrak{G}_2^*, \psi$) are captured in this generalization (along with other similar Σ-protocols):*

[12] We further elaborate on this in Remark 2.
[13] Uniform distribution over $\mathfrak{G}_1^*$.

(1) Guillou-Quisquater [GQ90] (e-roots in an RSA group) for which $\mathfrak{G}_1^* = \mathfrak{G}_2^* = \mathbb{Z}_n^*$ *for a semi-prime* $n = pq$, $C = [0, e)$ *and* $\psi(w) \coloneqq w^e$ *for some prime* $e \in \mathbb{N}$.

(2) Schnorr [Sch90] (knowledge of discrete log): for which $\mathfrak{G}_1^* = \mathbb{Z}_{|\mathbb{G}|}^+, \mathfrak{G}_2^* = \mathbb{G}$ *where* $\mathbb{G}$ *is a cyclic group of prime order* $|\mathbb{G}|$, $C = [0, |\mathbb{G}|)$ *and* $\psi(w) \coloneqq g^w$ *for some* $g \in \mathbb{G}$.

(3) Chaum-Pedersen [CP93] (equality of discrete log): for which $\mathfrak{G}_1^* = \mathbb{Z}_{|\mathbb{G}|}^+, \mathfrak{G}_2^* = \mathbb{G} \times \mathbb{G}$ *where* $\mathbb{G}$ *is a cyclic group of prime order* $|\mathbb{G}|$, $C = [0, |\mathbb{G}|)$ *and* $\psi : \mathbb{Z}_{|\mathbb{G}|} \to \mathbb{G} \times \mathbb{G}$, $\psi(w) \coloneqq (g_1^w, g_2^w)$ *for* $g_1, g_2 \in \mathbb{G}$.

(4) Attema-Cramer [AC20] (opening of linear forms): for which $\mathfrak{G}_1^* = \mathbb{Z}_{|\mathbb{G}|}^\ell \times \mathbb{Z}_{|\mathbb{G}|}$, $\mathfrak{G}_2^* = (\mathbb{Z}_{|\mathbb{G}|}, \mathbb{G})$, $C = [0, |\mathbb{G}|)$ *and* $\psi((\mathbf{x}, \gamma)) \coloneqq (L(\mathbf{x}), \mathbf{g}^{\mathbf{x}} h^\gamma)$ *for some linear form* $L(\mathbf{x}) = \langle \mathbf{x}, \mathbf{s} \rangle$, $\mathbf{s} \in \mathbb{Z}_{|\mathbb{G}|}^\ell$

We also show that a variant of Blum's classic 3 move protocol [Blu87] for graph Hamiltonicity is stackable in Appendix A. This is not surprising, since in the third round the prover either: (1) opens a Hamiltonian path in a permutation of the graph. (2) provides the randomness for the commitment to the permuted adjacency matrix. In either case the distribution of third round messages only depends on the number of vertices in the Hamiltonian graph: either a cycle of n vertices or n^2 openings of commitments (random strings).

Lemma 2. *Blum's* Σ*-protocol [Blu87] is stackable.*

The proof of Lemma 2 can be found in Appendix A

6.3 Examples of Stackable "MPC-in-the-Head" Σ-Protocols

We now proceed to show that many natural "MPC-in-the-head" style [IKOS07] Σ-protocols (with minor modifications) are stackable. MPC-in-the-head (henceforth refereed to as IKOS) is a technique used for designing three-round, public-coin, zero-knowledge proofs using MPC protocols. At a high level, the prover emulates execution of an n-party MPC protocol Π virtually, on the relation function $\mathcal{R}(x, \cdot)$ using the witness w as input of the parties, and commits to the views of each party. An honest verifier then selects a random subset of the views to be opened and verifies that those views are consistent with each other and with an honest execution, where the output of Π is 1.

Achieving EHVZK. Since the first round messages in such protocols only consist of commitments to the views of all virtual partials, a subset of which are opened in the third round, a natural simulation strategy when proving zero-knowledge of such protocols is the following: (1) based on the challenge message, determine the subset of parties whose views will need be opened later, (2) imagining these as the "corrupt" parties, use the simulator of the MPC protocol to simulate their views, and, finally, (3) compute commitments to these simulated views for this subset of the parties and commitments to garbage values for the remaining virtual parties. Clearly, since the first round messages in this simulation strategy are computed after the third round messages, *these protocols are naturally EHVZK.*

Achieving recyclable third messages. To show that these Σ-protocols have recyclable third messages, we observe that in many MPC protocols, an *adversary's view can often be condensed and decoupled from the structure of the functionality/circuit* being evaluated. We elaborate this point with the help of an example protocol — semi-honest BGW [BGW88].

Recall that in the BGW protocol, parties evaluate the circuit in a gate-by-gate fashion on secret shared inputs[14] as follows: (1) for addition gates, the parties locally add their own shares for the incoming wire values to obtain shares of the outgoing wire values. (2) For multiplication gates, the parties first locally multiply their own shares for the incoming wire values and then secret share these multiplied share amongst the other parties. Each party then locally reconstructs these "shares of shares" to obtain shares of the outgoing wire values. (3) Finally, the parties reveal their shares for all the output wires in the circuit to all other parties and reconstruct the output.

[14]These shares are computed using some threshold secret sharing scheme, e.g., Shamir's polynomial based secret sharing [Sha79].

By definition, the view of an adversary in any semi-honest MPC protocol is indistinguishable from a view simulated by the simulator with access to the corrupt party's inputs and the protocol output. Therefore, to understand the view of an adversary in this protocol, we recall the simulation strategy used in this protocol:

1. For each multiplication gate in the circuit, the simulator sends random values on behalf of the honest parties to each of the corrupt parties.

2. For the output wires, based on the messages sent to the adversary in the previous step and the circuit that the parties are evaluating, the simulator first computes the messages that the corrupt parties are expected to send to the honest parties. It then uses these messages and the output of protocol to simulate the messages sent by the honest parties to the adversary. Recall that this can be done because these messages correspond to the shares of these parties for the output wire values, and in a threshold secret sharing scheme, the shares of an adversary and the secret, uniquely define the shares of the remaining parties.

Observe that the computation done by the simulator in the first part is independent of the actual circuit or function being computed (it only depends on the number of multiplication gates in the circuit). We refer to the messages computed in (1) and the inputs of the corrupt parties as the *condensed view* of the adversary. Additionally, given these simulated views, the output of the protocol, and the circuit/functionality, the simulated messages of the honest parties in (2) can be computed deterministically. Looking ahead, because the output of relation circuits — the circuits we are interested in simulating — should always be 1 to convince the verifier, this deterministic computation will be straight forward. Since the condensed view is not dependent on the function being computed, it can be used with "any" functionality in the second step to compute the remaining view of the adversary. In other words, given two arithmetic circuits with the same number of multiplication gates, the condensed views of the adversary in an execution of the BGW protocol for one of the circuits can be re-interpreted as their views in an execution for the other one. We note that circuits can always be "padded" to be the same size, so this property holds more generally.

As a result, for IKOS-style protocols based on such MPC protocols, while some strict structure must be imposed upon third round messages (which are views of a subset of virtual parties) when *verifying* that they have been generated correctly, the third round messages themselves can simply consist of these condensed views (and not correspond to any particular functionality) and hence can be re-used. To make this work, we must make a slight modification to the IKOS compiler. As before, in the first round, the prover will commit to the views (where they are associated with a given function f) of all parties in the first round. However, in the third round, the prover can simply send the condensed views of the opened parties to the verifier. The verifier can deterministically compute the remaining view of these parties w.r.t. the appropriate relation function f and check if they are consistent amongst each other and with the commitments sent in the first round. Since the third round messages in this protocol are not associated with any function, it is now easy to see that they can be the distribution of these messages is independent of the instance.

Building on this intuition, we show that many natural MPC protocols produce stackable Σ-protocols for circuits of the same size when used with the IKOS compiler. Before giving a formal description of the required MPC property, we recall the IKOS compiler in more detail, assuming that the underlying MPC protocol has the following three-functions associated with it: ExecuteMPC emulates execution of the protocol on a given function with virtual parties and outputs the actual views of the parties, CondenseViews takes the views of a subset of the parties as input and outputs their condensed views, and ExpandViews takes the condensed views of a subset of the parties and returns their actual view w.r.t. a particular function.

IKOS Compiler. Let $f = \mathcal{R}(x, \cdot)$. In the first round, the prover runs ExecuteMPC on f and the witness w to obtain views of the parties and commits to each of these views. In the second round, the verifier samples a random subset of parties as its challenge message. Size of this subset is equal to the maximal corruption threshold of the MPC protocol. In the third round, the prover uses CondenseViews to obtain condensed views for this subset of parties and sends them to the verifier along with the randomness used to commit to the original views of these parties in the first round. The verifier runs ExpandViews on f and the condensed views received in the third round to obtain the corresponding original views. It checks if these are consistent

with each other and are valid openings to commitments sent in the first round. Depending on the corruption threshold and the security achieved by the underlying MPC protocol, the above steps might be repeated a number of times to reduce the soundness error. Below we restate the main theorem from [IKOS07], which also trivially holds for our modified variant.

Theorem 3 (IKOS [IKOS07])**.** *Let $\mathcal{L}$ be an NP language, $\mathcal{R}$ be its associated NP-relation and $\mathcal{F}$ be the function set $\{\mathcal{R}(x,\cdot) : \forall x \in \mathcal{L}\}$. Assuming the existence of non-interactive commitments, the above compiler transforms any MPC protocol for functions in $\mathcal{F}$ into a Σ-protocol for the relation $\mathcal{R}$.*

Next, we formalize the main property of MPC protocols that facilitates in achieving recyclable third messages when compiled with the above IKOS compiler. We characterize this property w.r.t. a function set $\mathcal{F}$, and require the MPC protocol to be such that the condensed views can be expanded for any $f \in \mathcal{F}$. For our purposes, it would suffice, even if the condensed view of the adversary is dependent on the final output of the protocol, as long as it is independent of the functionality. This is because, in our context, the circuit being evaluated will be a relation circuit with the statement hard-coded and should always output 1 in order to convince the verifier.

Definition 10 ($\mathcal{F}$-universally simulatable MPC)**.** *Let Π be an n-party MPC protocol that is capable of securely computing any function $f \in \mathcal{F}$ (where $\mathcal{F} : \mathcal{X}^n \to \mathcal{O}$) against any semi-honest adversary $\mathcal{A}$ who corrupts a set $\mathcal{I} \subset [n]$ of parties, such that $\mathcal{I} \in \mathcal{C}$, where $\mathcal{C}$ is the set of admissible corruption sets. We say that Π is $\mathcal{F}$-universally simulatable if there exists a 3-tuple of PPT functions* $(\mathsf{ExecuteMPC}, \mathsf{ExpandViews}, \mathsf{CondenseViews})$ *and a non-uniform PPT simulator $\mathcal{S}^{\text{F-MPC}} : \mathcal{F} \times \mathcal{C} \times \mathcal{O} \to V^*$, defined as follows*

- $(\{\mathsf{view}_i\}_{i\in[n]}, o) \leftarrow \mathsf{ExecuteMPC}(f, \{x_i\}_{i\in[n]})$*: This function takes inputs of the parties $\{x_i\}_{i\in[n]} \in \mathcal{X}^n$ and a function $f \in \mathcal{F}$ as input and returns the views $\{\mathsf{view}_i\}_{i\in[n]}$ of all parties and their output $o \in \mathcal{O}$ in protocol Π.*
- $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}} \leftarrow \mathsf{CondenseViews}(f, \mathcal{I}, \{\mathsf{view}_i\}_{i\in\mathcal{I}}, o)$*: This function takes as input the set of corrupt parties $\mathcal{I} \in \mathcal{C}$, views of the corrupt parties $\{\mathsf{view}_i\}_{i\in\mathcal{I}}$ and the output of the protocol $o \in \mathcal{O}$ and returns their condensed views $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}}$.*
- $\{\mathsf{view}_i\}_{i\in\mathcal{I}} \leftarrow \mathsf{ExpandViews}(f, \mathcal{I}, \{\mathsf{con.view}_i\}_{i\in\mathcal{I}}, o)$*: This function takes as input the functionality $f \in \mathcal{F}$, set of corrupt parties $\mathcal{I} \in \mathcal{C}$, condensed views $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}}$ of the corrupt parties and the output of the protocol $o \in \mathcal{O}$ and returns their views $\{\mathsf{view}_i\}_{i\in\mathcal{I}}$.*
- $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}} \leftarrow \mathcal{S}^{\text{F-MPC}}(f, \mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, o)$*: The simulator takes as input the functionality $f \in \mathcal{F}$, set of corrupt parties $\mathcal{I} \in \mathcal{C}$, inputs of the corrupt parties $\{x_i\}_{i\in\mathcal{I}} \in \mathcal{X}^{|\mathcal{I}|}$ and the output of the protocol $o \in \mathcal{O}$ and returns simulated condensed views $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}}$ of the corrupt parties.*

And these functions satisfy the following properties:

1. ***Condensing-Expanding Views is Deterministic:*** *For all $\{x_i\}_{i\in[n]} \in \mathcal{X}^n$ and $\forall f \in \mathcal{F}$, let $(\{\mathsf{view}_i\}_{i\in[n]}, o) \leftarrow \mathsf{ExecuteMPC}(f, \{x_i\}_{i\in[n]})$. For all $\mathcal{I} \in \mathcal{C}$ it holds that:*

$$\Pr\left[\mathsf{ExpandViews}(f, \mathcal{I}, \mathsf{CondenseViews}(f, \mathcal{I}, \{\mathsf{view}_i\}_{i\in\mathcal{I}}, o), o) = \{\mathsf{view}_i\}_{i\in\mathcal{I}}\right] = 1$$

2. ***Indistinguishability of Simulated Views from real execution:*** *For all $\{x_i\}_{i\in[n]} \in \mathcal{X}^n$ and $\forall f \in \mathcal{F}$, let $(\{\mathsf{view}_i\}_{i\in[n]}, o) \leftarrow \mathsf{ExecuteMPC}(f, \{x_i\}_{i\in[n]})$. For all $\mathcal{I} \in \mathcal{C}$ it holds that:*

$$\mathsf{CondenseViews}(f, \mathcal{I}, \{\mathsf{view}_i\}_{i\in\mathcal{I}}, o) \approx \mathcal{S}^{\text{F-MPC}}(f, \mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, o)$$

3. ***Indistinguishability of Simulated Views for all functions:*** *For any $\mathcal{I} \in \mathcal{C}$, all inputs $\{x_i\}_{i\in\mathcal{I}} \in \mathcal{X}^{|\mathcal{I}|}$ of the corrupt parties, and all outputs $o \in \mathcal{O}$, there exists a function-independent distribution $\mathcal{D}_{\{x_i\}_{i\in\mathcal{I}}, o}$, such that $\forall f \in \mathcal{F}$, if $\exists \{x_i\}_{i\in[n]\setminus\mathcal{I}}$ for which $f(\{x_i\}_{i\in[n]\setminus\mathcal{I}}, \{x_i\}_{i\in\mathcal{I}}) = o$, then it holds that:*

$$\mathcal{D}_{\{x_i\}_{i\in\mathcal{I}}, o} \approx \mathcal{S}^{\text{F-MPC}}(f, \mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, o)$$

We note that a central notion used in the "stacked-garbling literature" (for communication efficient disjunction for garbled circuit based zero-knowledge proofs) is a special case of $\mathcal{F}$-universally simulatable:

Remark 4 (Topology Decoupled Garbled Circuits and $\mathcal{F}$-universally simulatable MPC.)**.** *The notion of topology decoupled garbled circuits introduced by Kolesnikov [Kol18] is a special case of $\mathcal{F}$-universally simulatable MPC: a topology decoupled garbled circuit (E, T) separates the cryptographic material (E, e.g. garbling tables) and topology (T, i.e. wiring) of a garbled circuit and (informally stated) requires that generating E for different topologies introduces indistinguishable distributions. Letting X be the garbled input labels[15] held by the evaluator, in $\mathcal{F}$-universally simulatable terminology (E, X) would constitute the "condensed view", while (E, X, T)[16] would constitute the "expanded views" , indistinguishablilty of simulated views for functions with the same number of gates and inputs follows easily from the "topology decoupling" of the garbled circuits and the uniform distribution of the input labels.*

We now proceed to show that when instantiated with an $\mathcal{F}$-universally simulatable MPC protocol, Theorem 3 yields a stackable Σ-protocol for languages with relation circuits in $\mathcal{F}$.

Theorem 4 ($\mathcal{F}$-universally simulatable implies stackable)**.** *The IKOS compiler (see Theorem 3) yields an stackable Σ-protocol for languages with relation circuit in $\mathcal{F}$ when instantiated with an $\mathcal{F}$-universally simulatable MPC protocol (see Definition 10) with privacy and robustness (See Definitions 2,3) against a subset of the parties.*

Proof. We define the distribution $\mathcal{D}^{(z)}_{x,c}$, where $c \in \{0,1\}^\kappa$ describes a set of players $\mathcal{I} \in \mathcal{C}$ as follows:

$$\mathcal{D}^{(z)}_{x,c} = \left\{ \{\mathsf{con.view}_i, r_i\}_{i\in\mathcal{I}} \mid \{\mathsf{con.view}_i\}_{i\in\mathcal{I}} \xleftarrow{\$} \mathcal{D}_{\{x_i\}_{i\in\mathcal{I}},1}, \{r_i\}_{i\in\mathcal{I}} \xleftarrow{\$} \{0,1\}^{\mathcal{I}\cdot\lambda} \right\}.$$

The EHVZK simulator (derived from the standard IKOS simulator) $\mathcal{S}^{\text{EHVZK}}(1^\lambda, f, c, z)$ takes a description $f \in \mathcal{F}$ and challenge $c \in \{0,1\}^\kappa$ describing a set of players $\mathcal{I} \in \mathcal{C}$, and third round message $z = \{\mathsf{con.view}_i, r_i\}_{i\in\mathcal{I}} \xleftarrow{\$} \mathcal{D}^{(z)}_{x,c}$ and computes the first round message as follows:

It runs $\mathsf{ExpandViews}(f, \mathcal{I}, \{\mathsf{con.view}_i\}_{i\in\mathcal{I}}, 1)$ to obtain original views $\{\mathsf{view}_i\}_{i\in\mathcal{I}}$. It then commits to these original views of the opened parties $\{\mathsf{com}_i = \mathsf{Com}(\mathsf{view}_i; r_i)\}_{i\in\mathcal{I}}$, and generates dummy commitments $\{\mathsf{com}_i = \mathsf{Com}(0; r_i)\}_{i\in[n]\setminus\mathcal{I}}$ for the views of the remaining parties, using some additional randomness $\{r_i\}_{i\in[n]\setminus\mathcal{I}}$. It returns first round message $a = \{\mathsf{com}_i\}_{i\in[n]}$.

We now argue indistinguishability between a real transcript and the above simulated transcript. Let $\mathcal{H}_0$ be the distribution over a real transcript and $\mathcal{H}_3$ be the above simulated transcript. We define the following intermediate hybrids:

- $\mathcal{H}_1$: Compute dummy commitments instead of honest ones for the unopened players in the first round. Indistinguishability between $\mathcal{H}_0$ and $\mathcal{H}_1$ follows from the hiding property of commitments.
- $\mathcal{H}_2$: Instead of honestly computing $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}}$, sample these condensed views from $\mathcal{S}^{\text{F-MPC}}(f, \mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, 1)$ and use $\mathsf{ExpandViews}(f, \mathcal{I}, \{\mathsf{con.view}_i\}_{i\in\mathcal{I}}, 1)$ to obtain original views $\{\mathsf{view}_i\}_{i\in\mathcal{I}}$. Indistinguishability between $\mathcal{H}_2$ and $\mathcal{H}_3$ follows from indistinguishability of simulated views from real execution (See Definition 10) of the MPC protocol.
- $\mathcal{H}_3$: Instead of sampling $\{\mathsf{con.view}_i\}_{i\in\mathcal{I}}$ from $\mathcal{D}_{\{x_i\}_{i\in\mathcal{I}},1}$, sample these condensed views from $\mathcal{S}^{\text{F-MPC}}(f, \mathcal{I}, \{x_i\}_{i\in\mathcal{I}}, 1)$. Indistinguishability between $\mathcal{H}_2$ and $\mathcal{H}_3$ follows from indistinguishability of simulated views for all functions (See Definition 10) of the MPC protocol and from the fact that condensing-expanding views is a deterministic process.

From transitivity of computational indistinguishability, it follows that $\mathcal{H}_0 \approx \mathcal{H}_3$. Hence, this Σ-protocol achieves EHVZK. For recyclable third messages, we observe that since $\mathcal{D}^{(z)}_{x,c}$ as defined above does not

[15] Obtained using an oblivious transfer.
[16] Where T can be computed from f.

depend of the functionality of the MPC protocol, it is also independent of the statement, which in the IKOS compiler is hard-wired in the functionality. Therefore, $\mathcal{D}_c^{(z)}$ is the same as $\mathcal{D}_{x,c}^{(z)}$ and this protocol is indeed stackable.

□

We now use Theorem 4 to show that two popular IKOS-based Σ-protocols are stackable, namely KKW [KKW18] and Ligero [AHIV17]. The intuition is very similar to that presented for semi-honest BGW above — condensed views (which correspond to the third round messages sent in these protocols) can be used to simulate transcripts with respect to multiple functionalities. We formally state this with respect to functions of the same "size", but note circuits can always be padded to have the same size.

We prove the following Lemmas (and give a description of the underlying MPC protocol in KKW and Ligero) in Appendix D.1 and Appendix E respectively.

Lemma 3 (KKW [KKW18] is stackable)**.** *For any $m \in \mathbb{N}$, the underlaying MPC in KWW is $\mathcal{F}$-universally simulatable for $\mathcal{F}$ consisting of circuits with m multiplications.*

Lemma 4 (Ligero [AHIV17] is stackable)**.** *For any $m \in \mathbb{N}$, the underlaying MPC in Ligero is $\mathcal{F}$-universally simulatable for $\mathcal{F}$ consisting of circuits with m gates.*

6.4 Well-Behaved Simulators

As outlined in Section 3, a critical step of our compilation framework is applying the simulator of the underlying Σ-protocols to the inactive clauses. Note that these inactive clauses might not be true (if the language is non-trivial), even though the disjunction is satisfied, as such our framework should be applicable to the case where some of the instances are false.

We note, however, that the behavior of a simulator is only defined with respect to statements that are in the NP language — that is, true instances. As such, if the disjunction contains false clauses, there is no guarantee that the simulator will produce an accepting transcript. This would cause problems with verification — the verifier will know that one of the transcripts is not accepting, but will not know if this is due to a simulation failure or malicious prover. As such, we must carefully consider what simulators will produce when executed on a false instance.

As noted in [GO94], the simulators that are commonly constructed in most proofs of zero-knowledge will *usually* output accepting transcripts when executed on these false instances. If the simulator were able to consistently output non-accepting transcripts for false instances, it could be used to decide the NP language in polynomial time. However, it is possible to define a valid simulator that produces an output that is not an accepting transcript with non-negligible probability e.g. (1) the input instance is trivially false (*e.g.* a connected graph with 4 nodes is not 3-colorable), or (2) the simulator has a hard-coded set of false instances on which it deviates from its normal behavior. Indeed, a probabilistic simulator may also output a non-accepting transcript in each of these cases only occasionally, possibly depending on the challenge. Note that in both cases, a verifier will also be able to detect that the input instance is false simply by running the simulator themselves.

Looking ahead, if one of the underlying Σ-protocols has a simulator with this kind of logic, our compiled protocol could have a non-negligible *soundness* error proportional to the probability (over the random coins of the challenge) that the simulator outputs a non-accepting transcript. Producing of a non-accepting transcript in this way does not undermine zero-knowledge: simulation is only required for statements in the language. However the verifier would reject the proof of the disjunction by an honest prover, on the flip side, if the verification algorithm allows some transcript to be non-accepting, a malicious prover could trivially exploit this property to violate soundness. Therefore it is important for completeness that the simulator always produces accepting transcripts.

We emphasize that this is a corner case: commonly constructed simulators will produce accepting transcripts even on false instances. Nevertheless, We observe that any Σ-protocol can be generically transformed into one that has a simulator that outputs accepting transcripts for all statements. We refer to such simulators as *well-behaved* simulators.

Definition 11 (Well-Behaved Simulator)**.** *We say that a Σ-protocol $\Sigma = (A, Z, \phi)$ for a* NP *language $\mathcal{L}$ and associated relation $\mathcal{R}(x, w)$ has a* well-behaved simulator *if the simulator $\mathcal{S}$ defined for Special Honest Zero-Knowledge has the following property: For any statement x (for both $x \in \mathcal{L}$ and $x \notin \mathcal{L}$),*

$$\Pr\left[\phi(x, a, c, z) = 1 \mid c \xleftarrow{\$} \{0,1\}^\lambda; a \leftarrow \mathcal{S}(x, c)\right] = 1$$

We say that an EHVZK Σ-protocol has a well-behaved simulator if its extended simulator $\mathcal{S}^{\text{EHVZK}}$ has the natural extension of this property.

We formally prove the following theorem in Appendix B.

Lemma 5 (Simulators are well-behaved without loss of generality)**.** *Every Σ-protocol Π can be converted to a Σ-protocol Π' for the same relation with a well-behaved simulator. Furthermore, if Π' is EHVZK then Π' is also EHVZK and if Π has recyclable third messages then Π' has recyclable third messages.*

In all subsequent sections, we assume w.l.o.g. that all Σ-protocols, have a well-behaved simulator and that is what we use in our compilers.

7 Self-Stacking: Disjunctions With The Same Protocol

We now present a self-stacking compiler for Σ-protocols, presented in Figure 5. By self-stacking, we mean a compiler that takes a *stackable* Σ-protocol Π for a language $\mathcal{L}$ and produces a Σ-protocol for language with disjunctive statements of the form $(x_1 \in \mathcal{L}) \vee (x_2 \in \mathcal{L}) \vee \ldots \vee (x_\ell \in \mathcal{L})$ with communication complexity proportional to the size of a single run of the underlying Σ-protocol (along with an additive factor that is linear in ℓ and λ). The key ingredient in our compiler is the partially-binding vector commitments (See Definition 4), which will allow the prover to efficiently compute verifying transcripts for the inactive clauses.

The compiler generates an accepting transcript (a_α, c, z^*) to the active clause $\alpha \in [\ell]$ using the witness, and then simulates accepting transcripts for each non-active clause, using the extended simulator. Recall that this extended simulator takes in a third round message z and a challenge c and produces a first round message a such that $\phi(x, a, c, z) = 1$. Thus, the prover can *re-use* the third round message z^*, for each simulated transcript, thereby reducing communication to the size of a single third round message. For a more detailed overview, we refer the reader to Section 3.

We now present a formal description of the self-stacking compiler:

Theorem 5 (Self-Stacking)**.** *Let $\Pi = (A, Z, \phi)$ be a stackable (See Definition 9) Σ-protocol for the NP relation $\mathcal{R} : \mathcal{X} \times \mathcal{W} \to \{0,1\}$ and let* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv}, \mathsf{BindCom})$ *be a 1-out-of-ℓ binding vector commitment scheme (See Definition 4). For any* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, the compiled protocol $\Pi' = (A', Z', \phi')$ described in Figure 5 is a stackable Σ-protocol for the relation $\mathcal{R}' : \mathcal{X}^\ell \times ([\ell] \times \mathcal{W}) \to \{0,1\}$, where $\mathcal{R}'((x_1, \ldots, x_\ell), (\alpha, w)) := \mathcal{R}(x_\alpha, w)$.*

Proof. We now prove that the protocol $\Pi' = (A', Z', \phi')$ described in Figure 5 is a stackable Σ-protocol for the relation $\mathcal{R}'((x_1, \ldots, x_\ell), (\alpha, w)) \coloneqq \mathcal{R}(x_\alpha, w)$.

Completeness. Completeness follows directly from the completeness of the underlying Σ-protocol and the commitment scheme. Note that because the underlying Σ-protocol has a well-behaved simulator, the prover will not produce non-accepting transcripts on any clauses embedding false instances.

Special Soundness. We create an extractor $\mathcal{E}'$ for the protocol Π' using the extractor $\mathcal{E}$ for the underlying Σ-protocol Π. The extractor $\mathcal{E}'$ is given two accepting transcripts for the protocol Π' that share a first round message, *i.e.* a, c, z, c', z'. The extractor uses this input to recover 2ℓ total transcripts (2 for each clause), $(a_i, c, z^*), (a'_i, c', z'^*)$ for $i \in [\ell]$. By the partial binding property of the partially-binding vector commitment scheme, with all but negligible probability there exists an $\alpha \in [\ell]$ such that $a_\alpha = a'_\alpha$. $\mathcal{E}'$ then invokes the extractor of Π on these transcripts to recover $w \leftarrow \mathcal{E}(1^\lambda, x_\alpha, a_\alpha, c_\alpha, z^*, c'_\alpha, z'^*)$ and returns (α, w). Because

the underlying extractor $\mathcal{E}$ cannot fail with non-negligable probability, the $\mathcal{E}'$ succeeds with overwhelming probability.

Extended Honest-Verifier Zero-Knowledge (and Recyclable Third Messages). For $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, let $\mathcal{D}_c^{(z)'} \coloneqq \{(\mathsf{ck}, r, z) \mid (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{1\}); r \xleftarrow{\$} \{0,1\}^\lambda; z \xleftarrow{\$} \mathcal{D}_c^{(z)}\}$ be the simulated distribution over third round messages for Π'. We construct the extended simulator for Π' by running the underlying extended simulating $\mathcal{S}^{\text{EHVZK}}$ for every clause and committing to the tuple of first round message $(a_1, \ldots, a_\ell)$ using commitment key ck and randomness r:

$$
\begin{array}{l}
a' \leftarrow \mathcal{S}^{\text{EHVZK}\prime}((x_1, \ldots, x_\ell), c, z' = (\mathsf{ck}, r, z)) \\
\hline
1: \quad \textbf{for } i \in [\ell]: \text{ Compute } a_i \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_i) \\
2: \quad \mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v} = (a_1, \ldots, a_\ell); r) \\
3: \quad \textbf{return } (\mathsf{ck}, \mathsf{com})
\end{array}
$$

Let $\mathcal{D}^{(\alpha,w)}$ denote the distribution of transcripts resulting from an honest prover possessing witness (α, w) running Π' with an honest verifier on the statement $(x_1, \ldots, x_\ell)$, where $\mathcal{D}^{(\alpha,w)}$ is over the randomness of the prover and the verifier. We now proceed using a hybrid argument. Let $\mathcal{H}^{(\alpha)}$ be the same as $\mathcal{D}^{(\alpha,w)}$, except let the first round message of clause α be generated by simulation, *i.e.* $a_\alpha \leftarrow \mathcal{S}^{\text{EHVZK}}(x_\alpha, c, z)$. By the EHVZK of Π, $\mathcal{H}^{(\alpha)} \approx \mathcal{D}^{(\alpha,w)}$. Next, let $\mathcal{H}^{(\alpha,\mathsf{ck})}$ be the same as $\mathcal{H}^{(\alpha)}$ except let the commitment key ck be generated with the binding position as $B = \{1\}$, *i.e.* $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{1\})$. Observe that $\mathcal{H}^{(\alpha,\mathsf{ck})} \stackrel{\text{p}}{=} \mathcal{H}^{(\alpha)}$ by the (perfect) hiding of the partially-binding commitment scheme. Lastly note that $\mathcal{H}^{(\alpha,\mathsf{ck})}$ matches the output distribution of $\mathcal{S}^{\text{EHVZK}\prime}((x_1, \ldots, x_\ell), c, \mathcal{D}_c^{(z)'})$.

Therefore Π' is a stackable Σ-protocol.

□

We now proceed to analyze the complexity of our resulting protocol.

Communication Complexity. Let $\mathtt{CC}(\Pi)$ be the communication complexity of Π. Then, the communication complexity of the Π' obtained from Theorem 5 is $(\mathtt{CC}(\Pi) + |\mathsf{ck}| + |\mathsf{com}| + |r'|)$, where the sizes of ck, com and r' depends on the choice of partially-binding vector commitment scheme and are independent of $\mathtt{CC}(\Pi)$. With our instantiation of partially binding vector commitments, the size of $|\mathsf{ck}|, |r'|$ will depend linearly on ℓ. However since our resulting protocol Π' is also stackable, the communication complexity can be reduced further to $\mathtt{CC}(\Pi) + 2\log(\ell)(|\mathsf{ck}| + |\mathsf{com}| + |r'|)$ by recursive application of the compiler as follows: let $\Pi_1 = \Pi$ and for $n > 1$ let Π_{2n} be the outcome of applying the compiler from Theorem 5 with $\ell = 2$ to Π_n. Note that Π_ℓ only applies the stacking compiler $\lceil \log(\ell) \rceil$ times and that $\mathtt{CC}(\Pi_{2n}) = \mathtt{CC}(\Pi_n) + |\mathsf{ck}| + |\mathsf{com}| + |r'|$. Therefore $\mathtt{CC}(\Pi_\ell) = \mathtt{CC}(\Pi) + 2\log(\ell)(|\mathsf{ck}| + |\mathsf{com}| + |r'|)$.

Computational Complexity. In general, the computation complexity of this protocol is ℓ times that of Π. However, in many protocols, the simulator is much faster than computing an honest transcript. We note that for such protocols, our compiler is expected to also get savings in the computation complexity.

7.1 Self Stacking for Instances in Multiple Languages

Many known constructions of Σ-protocols work for more than one language. For instance, most MPC-in-the-head style Σ-protocols (e.g. KKW [KKW18] , Ligero [AHIV17]) can support all languages with a polynomial sized relation circuit, as long as the underlying MPC protocol works for any polynomial sized function. However, because Σ-protocols are defined w.r.t. a particular NP language/relation, instantiating [KKW18] for two different NP languages $\mathcal{L}_1$ and $\mathcal{L}_2$ will (by definition) result in two distinct Σ-protocols. Therefore, applied naïvely, our compiler could only be used to stack Σ-protocols from [KKW18] for the exact same relation circuit.

We note that this seemingly artificial restriction can be relaxed and in many cases, allowing our compiler to stack Σ-protocols based on a particular *technique* for clauses of the form $(x_1 \in \mathcal{L}_1) \vee (x_2 \in \mathcal{L}_2) \vee \ldots \vee (x_\ell \in \mathcal{L}_\ell)$.

Self-Stacking Compiler

Statement: $x = x_1, \ldots, x_n$
Witness: $w = (\alpha, w_\alpha)$

- **First Round:** Prover computes $A'(x, w; r^p) \to a$ as follows:
  - Parse $r^p = (r^p_\alpha \| r)$.
  - Compute $a_\alpha \leftarrow A(x_\alpha, w_\alpha; r^p_\alpha)$.
  - Set $\mathbf{v} = (v_1, \ldots, v_\ell)$, where $v_\alpha = a_\alpha$ and $\forall i \in [\ell] \setminus \alpha$, $v_i = 0$.
  - Compute $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{\alpha\})$.
  - Compute $(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}; r)$.
  - Send $a = (\mathsf{ck}, \mathsf{com})$ to the verifier.
- **Second Round:** Verifier samples $c \leftarrow \{0,1\}^\lambda$ and sends it to the prover.
- **Third Round:** Prover computes $Z'(x, w, c; r^p) \to z$ as follows:
  - Parse $r^p = (r^p_\alpha \| r)$.
  - Compute $z^* \leftarrow Z(x_\alpha, w_\alpha, c; r^p_\alpha)$.
  - For $i \in [\ell]/\alpha$, compute $a_i \leftarrow \mathcal{S}^{\text{EHVZK}}(x_i, c, z^*)$.
  - Set $\mathbf{v}' = (a_1, \ldots, a_\ell)$
  - Compute $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', \mathsf{aux})$ (where aux can be regenerated with r).
  - Send $z = (\mathsf{ck}, z^*, r')$ to the verifier.
- **Verification:** Verifier computes $\phi'(x, a, c, z) \to b$ as follows:
  - Parse $a = (\mathsf{ck}, \mathsf{com})$ and $z = (\mathsf{ck}', z^*, r')$
  - Set $a_i \leftarrow \mathcal{S}^{\text{EHVZK}}(x_i, c, z^*)$
  - Set $\mathbf{v}' = (a_1, \ldots, a_\ell)$
  - Compute and return:

$$b = (\mathsf{ck} \stackrel{?}{=} \mathsf{ck}') \wedge \left(\mathsf{com} \stackrel{?}{=} \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}'; r')\right) \wedge \left(\bigwedge_{i \in [\ell]} \phi(x_i, a_i, c, z^*)\right)$$

Figure 5: A compiler for stacking multiple instances of a Σ-protocol.

By "Σ-protocols based on a particular technique", we mean (for instance) protocols based on [KKW18]. This can be done by working with a "meta-language" that covers all languages of interest and is supported by that technique. This could for example be an NP complete language, which would allow us to use our self-stacking compiler by first reducing each x_i to an instance of the NP-complete language. However, reducing to NP complete languages without care will often add the significant cost of performing an NP-reduction to the complexity of our compiler, which may no longer be efficient. In many cases, it is often possible to find the "most suitable" meta-language without compromising on the efficiency. For instance, for any MPC-in-the-head style Σ-protocol, this meta-language is as simple as circuit satisfiability for circuits of a given size (where this size is determined based on the language with the largest relation circuit). This can be easily achieved

with the help of padding, without incurring any additional overhead. This observation combined with the fact that simulation is deterministic in both KKW and Ligero, we get a protocol for general disjunctions, where the communication is the same as the communication for a single instance for the clause with the largest relation circuit and additive factor that depends on $\log(\ell)$ and the security parameter.

8 Cross-Stacking: Disjunctions with Different Protocols

In the previous section, we presented a compiler that facilitated stacking of the same Σ-protocol. We now extend these ideas to allow stacking of different Σ-protocols, *i.e.* statements of the form $(x_1 \in \mathcal{L}_1) \vee (x_2 \in \mathcal{L}_2) \vee \ldots \vee (x_\ell \in \mathcal{L}_\ell)$. This allows picking the "best" Σ-protocol for each clause and getting stacking as an afterthought. Note that we saw a limited version of achieving this in Section 7.1, where the Σ-protocols all shared the same techniques, using the meta-language approach. However, that idea crucially relied on the fact that there exists such a meta-language that is also supported by the Σ-protocol technique that we want to stack. To avoid this requirement, we now consider the more complex case where the Σ-protocols rely on different techniques.[17] For instance, we explore how to stack Ligero-based [AHIV17] Σ-protocols with KKW-based [KKW18] Σ-protocols despite their dissimilarity. We build intuition while exploring barriers in an incremental manner below before finally making precise the notion of *cross-stacking*.

8.1 Cross Simulatability

As discussed above, the simplest intuitive example of cross-stacking is one where for each challenge c, multiple Σ-protocols share the same distribution over last round messages $\mathcal{D}_c^{(z)}$. This is most clear when the Σ-protocols are derived from the same techniques. In this case, the techniques from the self-stacking compiler can be used directly. In Section 7.1 we used the "meta-language" approach for KKW-based [KKW18] Σ-protocols. We now consider another example using Schnorr-like protocols that does not require us to work with a meta-language:

Example 1 (Preimages of homomorphisms with the same domain [CD98])**.** *Recall the protocol Π_ψ of Cramer and Damgård described earlier. Any two instances of Π_{ψ_1} and Π_{ψ_2} for one-way homomorphisms $\psi_1 : \mathfrak{G}_1^* \to \mathfrak{G}_2^*$ and $\psi_2 : \mathfrak{G}_1^* \to \mathfrak{G}_3^*$ with the same domain $\mathfrak{G}_1^*$ can be stacked* as through *they were the same protocol using the self-stacking compiler: recall that for both Π_{ψ_1} and Π_{ψ_2}, $\mathcal{D}_c^{(z)}$ is the uniform distribution over $\mathfrak{G}_1^*$. Concrete examples include generalizations of the Chaum-Pedersen [CP93] Σ-protocol $\Pi_{DlogEq,\ell}$ (shown in Figure 6), for showing equality of discrete log: for any $(\ell, g_1, \ldots, g_\ell)$ the homomorphism $\psi_{g_1,\ldots,g_\ell} : \mathbb{Z}_{|\mathbb{G}|} \to \mathbb{G}^\ell$ defined as $\psi_{g_1,\ldots,g_\ell}(w) \coloneqq (g_1^w, \ldots, g_\ell^w)$, has the same domain $\mathbb{Z}_{|\mathbb{G}|}$ (different ranges).*

It is easy to see that the self-stacking compiler can be extended to different Σ-protocols that are essentially the same and explicitly share third round message distributions. However, there are many protocols that may appear to have different third round distributions that can still be directly stacked. This is possible when structured distributions have their structure removed, leaving behind a "bunch of bits" that can be re-interpreted in different ways. For example:

Example 2 (KKW over different commutative rings)**.** *Consider two KKW-based Σ-protocols. Π_1 is for a language with a relation circuit defined over the ring $\mathbb{F}_{2^k}$, while Π_2 is for a language with relation circuit over $\mathbb{Z}_{2^k}$. If elements of both $\mathbb{F}_{2^k}$ and $\mathbb{Z}_{2^k}$ are encoded as k-bit strings and the multiplicative complexity of the relation circuits are the same, the the bit-wise distribution of $\mathcal{D}_c^{(z)}$ for Π_1 and Π_2 is the same (see Figure 13). Therefore, Π_1 and Π_2 can be stacked* as though *they were the same protocol using the self-stacking compiler and their extended simulators will re-use the bit-wise encodings of elements of one ring as though they were bit-wise encodings of the other ring. This approach can be generalized to any pair of finite commutative rings*

[17] We note that this distinction between self-stacking and cross-stacking is not a firm, technical one, but rather a conceptual difference. Taking the meta-language approach described in Section 7.1 to stacking Σ-protocols based on differing techniques naturally leads to the question of how well transcripts with differing structures and distributions can be re-used. We highlight these questions in this section under the name cross-stacking.

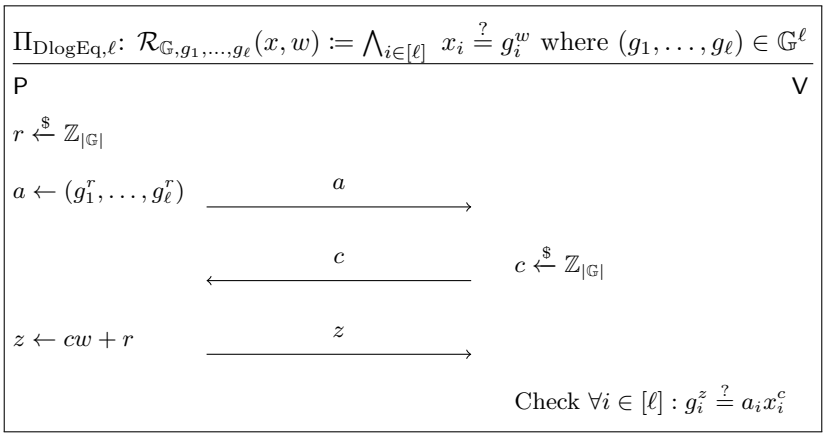

Figure 6: Generalized Chaum-Pedersen

R_1, R_2 st. the size of the rings differs by a constant multiplicative factor and the circuits are of the correct size. Specifically, if there exist a constant k such that $|R_1| = k|R_2|$ and the relation circuits are arithmetic circuits over R_1 and R_2 with multiplicative complexity m and $k \cdot m$ respectively.

Finally, we extend our ideas to stacking Σ-protocols with truly distinct $\mathcal{D}_c^{(z)}$. As re-use of third round messages is fundamental to our approach, the question becomes–*to what extent can the prover safely re-use the third round messages of different Σ-protocols?* In general, there are two considerations; some Σ-protocols may have uniquely long third round messages, letting the verifier identify which Σ-protocol was run honestly, and in some Σ-protocols, the third round messages may "not be long enough" to facilitate re-use. Conceptually, these two problems have the same fix: add bits to the third round messages of each Σ-protocol until their third-round message distributions are the same. The hope is that the number of bits shared by the third round message distributions of these protocols is large, so only a few bits need to be added.

We formalize this notion by considering a common "super distribution" $\mathcal{D}$ into which the third round messages of each Σ-protocol can be embedded and from which third round message for each Σ-protocol can be extracted. $\mathcal{D}$ represent the composite of the distributions of the third round messages of the Σ-protocols — parts of the distributions that can be re-used need not be duplicated, but elements unique to any given Σ-protocol are also included. We formalize the mapping between the distribution of third round messages and the super distribution $\mathcal{D}$ using a (possibly randomized) embedding function $F_{\Sigma\to\mathcal{D}}$ and a deterministic extraction function $\mathsf{TExt}_{\mathcal{D}\to\Pi}$. For example, $F_{\Pi\to\mathcal{D}}$ may add randomly sampled bits or cryptographic material to a third round message z in order to create an element $d \in \mathcal{D}$. $\mathsf{TExt}_{\mathcal{D}\to\Pi}$ might simply "select" the appropriate bits from d to construct z. We note that $\mathcal{D}$ is independent of c and therefore needs to cover all possible values of c. Thus, if $\mathcal{D}_c^{(z)}$ varies wildly across c for one of the Σ-protocols, $\mathcal{D}$ will need to be large. This, however, is not common in practice; for example, $\mathcal{D}_c^{(z)}$ is the same across all values of c for both KKW and Ligero. We now present the property that we will require for cross-stacking:

Definition 12 (Cross Simulatability)**.** *A stackable Σ-protocol $\Pi = (A, Z, \phi)$ is 'cross simulatable' w.r.t. a distribution $\mathcal{D}$ if there exists a PPT algorithm $F_{\Pi\to\mathcal{D}} : \mathcal{D}_c^{(z)} \to \mathcal{D}$ and a deterministic polynomial time algorithm $\mathsf{TExt}_{\mathcal{D}\to\Pi} : \{0,1\}^\kappa \times \mathcal{D} \to \mathcal{D}_c^{(z)}$ satisfying the following properties:*

Indistinguishable Embedding: *For all $c \in \{0,1\}^\kappa$:*

$$\mathcal{D} \approx \left\{ d \mid r \xleftarrow{\$} \{0,1\}^\lambda; z \xleftarrow{\$} \mathcal{D}_c^{(z)}; d \leftarrow F_{\Pi\to\mathcal{D}}(z;r) \right\}$$

Invertability: *For all* $c \in \{0,1\}^\kappa$, $z \in \mathcal{D}_c^{(z)}$ *and* $r \in \{0,1\}^\lambda$*:*

$$\mathsf{TExt}_{\mathcal{D}\to\Pi}(c, F_{\Pi\to\mathcal{D}}(z;r)) = z$$

We note that these two properties also directly imply that for all $c \in \{0,1\}^\kappa$*,*

$$\mathcal{D}_c^{(z)} \approx \left\{ z \mid d \xleftarrow{\$} \mathcal{D}; z \leftarrow \mathsf{TExt}_{\mathcal{D}\to\Pi}(c,d) \right\}$$

This property guarantees that third round messages in a Σ-protocol can be embedded into the (possibly larger) distribution $\mathcal{D}$, a generalization of Definition 8. Note that every stackable Σ-protocol is cross simulatable with its own third round message distribution. To make this property useful, we will require that a *set of* Σ-protocols are all cross simulatable with the *same* distribution $\mathcal{D}$. This property can be trivially satisfied by simply appending the distributions of the underlying stackable Σ-protocols, making $\mathcal{D}$ a tuple of elements of the underlying distributions; the challenge is to find small $\mathcal{D}$ for which this property holds.

With this definition in hand, we now show how Σ-protocols with very distinct features can be made cross simulatable with a distribution $\mathcal{D}$ that is very similar in size to the distributions over third round messages of these protocols using the example of KKW [KKW18] and Ligero [AHIV17]. This is despite the very distinct features of the two techniques: Ligero has negligible soundness error, players equal to the square-root of the multiplicative complexity of the circuit, and requires a sufficiently large field. KKW, on the other hand, has constant soundness that must be amplified, a constant number of players (independent of the circuit size), and operates over any commutative ring.

Example 3. *Consider a Ligero-based* Σ*-protocol* Π_1 *for a language with a relation circuit of size* C_1 *defined over the field* $\mathbb{F}_{2^k}$ *. Additionally, consider a KKW-based* Σ*-protocol* Π_2 *for a language with relation circuit with multiplicative complexity* C_2 *defined over the ring* $\mathbb{Z}_{2^k}$*.*

Recall that third round message in Ligero contain (1) commitments c_i *to the unopened players, and (2) a* $\sqrt{C_1}$ *sized set of field elements for each opened party (that are used for consistency checks). In KKW, third round messages contain (1) a punctured* PRF *seed that allows the verifier to check the preprocessing for correctness, (2) for each of the online phases that are opened, (a) a seed for each opened player, (b) a* $O(C_2)$ *set of bits to "correct" the preprocessing for one of the players, and (c) the broadcast messages of the unopened player (also of size* $O(C_2)$*). Let* $\mathcal{D}$ *be of the form*

$$\mathcal{D} = \{(c_1, \ldots, c_N, B) \mid \forall i \in [N] : c_i \leftarrow \mathsf{com}(\epsilon; r_i), r \xleftarrow{\$} \{0,1\}^\lambda, B \xleftarrow{\$} \{0,1\}^L\}$$

for some arbitrary values ϵ *and values* N *and* L *constants that depend on* C_1 *and* C_2 *and the choice of concrete parameters for the instantiated* Σ*-protocols.*

Both third round messages contain a large number of commitments that are never opened for the unopened parties. These can simply be re-used in $\mathcal{D}$*; Additionally, both protocols contain large sets of pseudorandom-looking bits: in Ligero, these take the form of field elements and in KKW these take the form of correction bits, broadcast messages, and a punctured* PRF *seed. Because these elements come from the same underlying* bit-wise *distribution, they can similarly be reused. However, the number of commitments and pseudorandom bits in each protocol may differ. As such,* $\mathcal{D}$ *contains the maximum number of commitments and pseudorandom bits from between the two protocols.*

Mapping into $\mathcal{D}$ *involves determining the size of the padding: if more commitments must be added, the mapping function samples arbitrary values* ϵ *and commits to them honestly. Note that these commitments will* never *be opened, so the contents do not matter. If more pseudorandom bits are required, the mapping function samples the required number of bits. Extracting a third round function involves selecting the appropriate number of commitments and pseudorandom bits and parsing these bits as needed. Note that if the sizes of* C_1 *and* C_2 *are appropriate (*$\sqrt{C_1} \approx C_2 \times$*(*number of repetitions*)), very little padding will be needed.*

Cross-Stacking Compiler

Statement: $x = x_1, \ldots, x_n$
Witness: $w = (\alpha, w_\alpha)$

- **First Round:** Prover computes $A'(x, w; r^p) \to a$ as follows:
  - Parse $r^p = (r^p_\alpha \| r \| r_{\text{map}})$.
  - Compute $a_\alpha \leftarrow A_\alpha(x_\alpha, w_\alpha; r^p_\alpha)$.
  - Set $\mathbf{v} = (v_1, \ldots, v_\ell)$, where $v_\alpha = a_\alpha$ and $\forall i \in [\ell] \setminus \alpha$, $v_i = 0$.
  - Compute $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{\alpha\})$.
  - Compute $(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}; r)$.
  - Send $a = (\mathsf{ck}, \mathsf{com})$ to Verifier.
- **Second Round:** Verifier samples $c \xleftarrow{\$} \{0,1\}^\kappa$ and sends it to Prover.
- **Third Round:** Prover computes $Z'(x, w_\alpha, c; r^p) \to z$ as follows:
  - Parse $r^p = (r^p_\alpha \| r \| r_{\text{map}})$.
  - Compute $z_\alpha \leftarrow Z(x_\alpha, w_\alpha, c; r^p_\alpha)$.
  - $d \leftarrow F_{\Pi_\alpha \to \mathcal{D}}(z_\alpha; r_{\text{map}})$
  - For $i \in [\ell]/\alpha$, compute
    - $z_i \leftarrow \mathsf{TExt}_i(c, d)$
    - $a_i \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_i)$
  - Set $\mathbf{v}' = (a_1, \ldots, a_\ell)$.
  - Compute $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', \mathsf{aux})$ (where aux can be regenerated with r)
  - Send $z = (\mathsf{ck}, d, r')$ to the verifier.
- **Verification:** Verifier computes $\phi'(x, a, c, z) \to b$ as follows:
  - Parse $a = (\mathsf{ck}, \mathsf{com})$ and $z = (\mathsf{ck}', d, r')$.
  - For $i \in [\ell]$, compute
    - $z_i \leftarrow \mathsf{TExt}_i(c, d)$
    - $a_i \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_i)$
  - Set $\mathbf{v}' = (a_1, \ldots, a_\ell)$
  - Compute and return

$$b = (\mathsf{ck} \stackrel{?}{=} \mathsf{ck}') \wedge \left(\mathsf{com} \stackrel{?}{=} \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}'; r')\right) \wedge \left(\bigwedge_{i \in [\ell]} \phi(x_i, a_i, c, z_i)\right)$$

  .

Figure 7: A compiler for stacking instances of multiple Σ-protocols.

8.2 Cross-Stacking from Cross Simulatability

With the definition of cross simulatability now in hand, we present our cross-stacking compiler. The approach is the same as the self-stacking compiler, but for a set of Σ-protocols cross simulatable with respect to the same $\mathcal{D}$.

Theorem 6 (Cross-Stacking)**.** *Let $\mathcal{D}$ be a distribution. For each $i \in [\ell]$, let $\Pi_i = (A_i, Z_i, \phi_i)$ be a stackable (See Definition 9) Σ-protocol for the NP relation $\mathcal{R}_i : \mathcal{X}_i \times \mathcal{W}_i \to \{0,1\}$, that is cross simulatable w.r.t. to a distribution $\mathcal{D}$, and let $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv})$ be a 1-out-of-ℓ binding vector commitment scheme (See Definition 4). For any $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, the protocol $\Pi' = (A', Z', \phi')$ described in Figure 7 is a <u>stackable</u> Σ-protocol for the relation $\mathcal{R}'((x_1, \ldots, x_\ell), (\alpha, w)) := \mathcal{R}_\alpha(x_\alpha, w)$.*

We give a proof of this theorem in Appendix C.

Complexity Analysis. Complexity of this protocol can be calculated in a similar manner as in the self-stacking compiler, except that here the distribution will depend on size of elements in $\mathcal{D}$ (let this be $|x_{\mathcal{D}}|$). Thus, the communication complexity of Π' is $(|x_{\mathcal{D}}| + \ell O(\lambda) + |\mathsf{com}| + |r'|)$. The impact of choosing any particular partially-binding vector commitment scheme remains the same. As before, the above compiler can be optimized further, to yield a protocol with communication complexity $(|x_{\mathcal{D}}| + 2\log(\ell)O(\lambda) + |\mathsf{com}| + |r'|)$, where $O(\lambda)$ can be minimized by using efficient constructions of partially-binding vector commitments, as shown in the complexity analysis of self-stacking.

9 k-out-of-ℓ Proofs of Partial Knowledge

We now briefly sketch how to efficiently (and generically) generalize the 1-of-ℓ technique in this paper to k-of-ℓ threshold proofs with communication complexity linear in k and logarithmic in ℓ. We note that the previous version of our paper had a different version of this construction where the communication complexity was linear in both k and ℓ, which is not optimal. In a follow-up work, Avitabile et al. [ABFV22] proposed an optimized construction for proofs of partial knowledge where the communication is linear in k and logarithmic in ℓ. Their work inspired us to observe a much simpler variant of our previous construction that also has a similar efficiency, which we discuss in this section. For reference, we include our old construction (with non-optimal communication) from the previous version of this paper in Appendix F.

A naïve attempt at turning a 1-of-ℓ proof into a threshold k-of-ℓ proof is to simply execute the 1-of-ℓ proof k times in parallel, this however is not sound: a cheating prover knowing just a single witness can simply prove satisfiability of the same clause in every execution. This can be avoided by forcing the prover to use a unique "tag" for every clause. Let $\mathcal{H}$ be a family of k universal hash functions $H : [\ell] \to \mathcal{D}$ with $|\mathcal{D}| \geq k$, at a high-level the construction proceeds as follows:

1. Prover samples a k-universal hash function $H \xleftarrow{\$} \mathcal{H}$ subject to $|\{H(\alpha)\}_{\alpha \in \mathscr{A}}| = k$.
2. Prover commits to the function: $\mathsf{com} \leftarrow \mathsf{Commit}(H; r)$.
3. The prover sends $\mathsf{com}, t_1 = H(\alpha_1), \ldots, t_k = H(\alpha_k)$ to the verifier and for every $i \in [k]$ proves:

$$
\begin{aligned}
&(\mathsf{com} = \mathsf{Commit}(H; r) \wedge t_i = H(1) \wedge (x_1, w) \in \mathcal{R}_1) \\
\vee\ &(\mathsf{com} = \mathsf{Commit}(H; r) \wedge t_i = H(2) \wedge (x_2, w) \in \mathcal{R}_2) \\
\vee\ &(\mathsf{com} = \mathsf{Commit}(H; r) \wedge t_i = H(3) \wedge (x_3, w) \in \mathcal{R}_3) \\
\vee\ &\ldots \\
\vee\ &(\mathsf{com} = \mathsf{Commit}(H; r) \wedge t_i = H(\ell) \wedge (x_\ell, w) \in \mathcal{R}_\ell)
\end{aligned}
$$

   Using the disjunction technique (e.g. the technique covered in this paper). In addition to checking the validity of each proof the verifier now additionally checks that every t_i is distinct.

Note that the compiler does not increase round-complexity as $\mathsf{com}, t_1, \ldots, t_k$ can be send in parallel with the disjunction proof. Soundness follows from the soundness of the 1-of-ℓ proof and the binding of the commitment scheme: intuitively if a malicious prover attempts to prove the same clause α_i multiple times then $H(\alpha_i)$ must be appear multiple times, otherwise the malicious prover would be able to open com to a commitment to H and H' where $H(\alpha_i) \neq H'(\alpha_i)$ and hence $H \neq H'$ which violates binding of the commitment. The precise soundness definition perfect/statistical/computational is inherited directly from the commitment scheme and proof system. Intuitively the proof above is zero-knowledge since for any set of $t_1, \ldots, t_k \in \mathcal{D} : \Pr_{H \xleftarrow{\$} \mathcal{H}}[t_1 = H(\alpha_1), \ldots, t_k = H(\alpha_k)]$ is independent of α by the definition of k universality by combining this with hiding of the commitment scheme and zero-knowledge the disjunction proof: the simulator works by setting $\forall i \in [k] : \alpha_i = i$, sampling H like the honest prover, then running the simulator of the disjunction proof for every i.

The primary challenge in the compiler above is to instantiate Commit and $\mathcal{H}$ such that $\mathsf{com} = \mathsf{Commit}(H; r) \wedge t_i = H(i)$ can be efficiently proven using a Sigma protocol. We instantiate Commit using a scheme which enables commitment to polynomials of degree $k-1$ (a family of k-universal hash functions), note that we do not require a 'polynomial commitment scheme' as commonly defined: in particular we do not require that $|\mathsf{com}|$ is succinct or the opening proofs have poly-logarithmic time/communication in the degree of the polynomial. The scheme is a natural one based on a linearly homomorphic commitment scheme:

1. The prover commits to $H(X) = \sum_{i=1}^{k} c_i \cdot X^{i-1}$ by committing to each coefficient individually, for all $i \in 1, \ldots, k$ commit to c_i using a homomorphic commitment $[c_i] \leftarrow \mathsf{Commit}_{\mathsf{Homo}}(c_i; r_i)$ and form $\mathsf{com} = ([c_1], \ldots, [c_k])$.
2. To open H at x, both parties homomorphically compute $[H(x)] = \sum_{i=1}^{k} x^i \cdot [c_k]$
3. The prover applies a zero-knowledge proof to show that $[H(x)]$ opens to y, without revealing the randomness of the commitment.

For completeness, the linearly homomorphic commitment scheme can be instanced with Pedersen commitments and the (honest-verifier) zero-knowledge proof with a generalization of a Schnorr proof to prove the opening of the Pedersen commitment; as shown earlier such a proof is stackable.

10 Measuring Concrete Efficiency

Although the efficiency of our compiler is self evident from the construction of the commitment scheme, we also include two measurements to demonstrate the efficiency more clearly. Specifically, we compare the impact of applying our self-stacking compiler to instance instances of the KKW Σ-protocol and constructing ring signatures by applying our self-stacking compiler to Schnorr signatures.

Self-Stacking KKW [KKW18]. Our first measurement that demonstrates the concrete efficiency of our compiler is self-stacking KKW [KKW18]. We compare the results of this protocol to the naïve approach of simply applying CDS [CDS94] to the equivalent disjunctive statement. Specifically, we compare our compiler and CDS applied applied to circuits containing 1000 and 100,000 multiplication gates and sweep between 1 and 1000 clauses. The results of this comparison can be found in Figure 8.

To compute this table, we compute the communication complexity of KKW for 128-bits ($\lambda = 128$) of classical (non-quantum) security. The communication complexity for ℓ clauses for our work and CDS are computed as follows:

$$\begin{aligned}
\mathsf{Size\text{-}KKW} &= 2\lambda + \tau \cdot \log \frac{M}{\tau} \cdot 3\lambda + \tau(\lambda \log n + 2m + |w| + 3\lambda), \\
\mathsf{Size\text{-}Stacked} &= \mathsf{Size\text{-}KKW} + 4\lambda \cdot \lceil \log \ell \rceil, \\
\mathsf{Size\text{-}CDS} &= \ell \cdot (\mathsf{Size\text{-}KKW} + \lambda),
\end{aligned}$$

Multiplications (m)	#Clauses (ℓ)	Comm. (CDS [CDS94])	Comm. (Stacked Σ, ours)
1000	1	14.6 KB	
1000	10	146.1 KB	14.9 KB
1000	100	1,461.4 KB	15.1 KB
1000	1000	1,461.4 KB	15.3 KB
100 000	1	583.9 KB	
100 000	10	5,838.6 KB	584.1 KB
100 000	100	58,386.4 KB	584.3 KB
100 000	1000	583,864.0 KB	584.5 KB

Figure 8: Concrete communications complexity for disjunctions over Boolean circuits ($R = \mathbb{F}_2$) with different multiplicative complexity, targeting 128-bits of security: $n = 64, M = 631, \tau = 23$. The communication complexity of our work is computed when recursive stacking is applied using the optimized commitment scheme described in Appendix G.

Ring Size (n)	Time (t)	Signature Size ($\lvert\sigma\rvert$)
2^1	4 ms	128 B
2^2	8 ms	192 B
2^3	12 ms	256 B
2^4	18 ms	320 B
2^5	24 ms	384 B
2^6	34 ms	448 B
2^7	50 ms	512 B
2^8	76 ms	576 B
2^9	127 ms	640 B
2^{10}	224 ms	704 B
2^{11}	414 ms	768 B
2^{12}	813 ms	832 B

Figure 9: Performance of ring signatures for rings of different sizes.
All benchmarks run on a single core of `AMD EPYC 7601` @ 2.2 GHz.

where Size-KKW and parameters (M, τ) are derived in KKW [KKW18].

Ring-signatures From Schnorr Signatures [Sch90]. Our second efficiency measurment is constructing ring-signatures [RST01] by applying our compiler to classical Schnorr Signatures. Specifically, we recursively apply the compiler from Theorem 5 (with the partially binding vector commitments from Figure 17), to the Schnorr [Sch90] identification protocol over the Ristretto group of `Curve25519` [Ber06]. Then, we apply the Fiat-Shamir [FS87] heuristic to obtain a signature of knowledge in the random oracle model. We implement the compiled protocol in the Rust programming language; our implementation is open source and is available at `https://github.com/rot256/research-stacksig` .

The concrete size of these ring signatures with 128 bits of security and n parties is $|\sigma| = 64 \cdot \lceil \log_2(n) \rceil + 64$ bytes. We present running times and signatures sizes for these ring signatures in Figure 9.[18]

Acknowledgments

We would like to thank the anonymous reviewers of CRYPTO 2021 for their helpful comments on our initial construction of the partially-binding commitments. Additionally, we would like to thank Nicholas Spooner

[18]The benchmarks can be reproduced by running `cargo bench` using a `nightly` Rust.

for his helpful comments on the definition of these commitments. Finally, we would like to thank Gennaro Avitabile, Vincenzo Botta, Daniele Friolo and Ivan Visconti, who in their follow-up work "Efficient Proofs of Knowledge for Threshold Relations" [ABFV22] pointed out some subtle definitional issues in the previous version of this paper and constructed an optimized threshold version of our compiler. This motivated us to observe a slightly different (and improved) variant of our threshold construction, which we describe in this updated version.

The first and second authors are supported in part by NSF under awards CNS-1653110, and CNS-1801479, and the Office of Naval Research under contract N00014-19-1-2292. The first author is also supported in part by NSF CNS grant 1814919, NSF CAREER award 1942789 and the Johns Hopkins University Catalyst award. The second author is also funded by DARPA under Contract No. HR001120C0084, as well as a Security and Privacy research award from Google. The third author is funded by Concordium Blockhain Research Center, Aarhus University, Denmark. The forth author is supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project and is supported by DARPA under Agreements No. HR00112020021 and Agreements No. HR001120C0084. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[ABFV22] Gennaro Avitabile, Vincenzo Botta, Daniele Friolo, and Ivan Visconti. Efficient proofs of knowledge for threshold relations. Cryptology ePrint Archive, Report 2022/746, 2022. `https://eprint.iacr.org/2022/746`.

[AC20] Thomas Attema and Ronald Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 513–543. Springer, Heidelberg, August 2020.

[ACF20] Thomas Attema, Ronald Cramer, and Serge Fehr. Compressing proofs of k-out-of-n partial knowledge. 2020. `https://eprint.iacr.org/2020/753`.

[ACK21] Thomas Attema, Ronald Cramer, and Lisa Kohl. A compressed Σ-protocol theory for lattices. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 549–579, Virtual Event, August 2021. Springer, Heidelberg.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[AOS02] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. 1-out-of-n signatures from a variety of keys. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 415–432. Springer, Heidelberg, December 2002.

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

[BCC+15] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, Jens Groth, and Christophe Petit. Short accountable ring signatures based on DDH. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *ESORICS 2015, Part I*, volume 9326 of *LNCS*, pages 243–265. Springer, Heidelberg, September 2015.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

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

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013.

[Blu87] Manuel Blum. How to prove a theorem so no one else can claim it. pages 1444–1451, 1987.

[BMRS20] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. `https://eprint.iacr.org/2020/1410`.

[CD98] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 424–441. Springer, Heidelberg, August 1998.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 174–187. Springer, Heidelberg, August 1994.

[CP93] David Chaum and Torben P. Pedersen. Transferred cash grows in size. In Rainer A. Rueppel, editor, *EUROCRYPT'92*, volume 658 of *LNCS*, pages 390–407. Springer, Heidelberg, May 1993.

[CPS+16] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Online/offline OR composition of sigma protocols. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 63–92. Springer, Heidelberg, May 2016.

[FLPS21] Prastudy Fauzi, Helger Lipmaa, Zaira Pindado, and Janno Siim. Somewhere statistically binding commitment schemes with applications. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 436–456, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[GK15] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, Heidelberg, April 2015.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract). In *27th FOCS*, pages 174–187. IEEE Computer Society Press, October 1986.

[GMY03] Juan A. Garay, Philip D. MacKenzie, and Ke Yang. Strengthening zero-knowledge protocols using signatures. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 177–194. Springer, Heidelberg, May 2003.

[GO94] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994.

[GQ90] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 216–231. Springer, Heidelberg, August 1990.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[HK20a] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Heidelberg, August 2020.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[JM20] Aram Jivanyan and Tigran Mamikonyan. Hierarchical one-out-of-many proofs with applications to blockchain privacy and ring signatures. *2020 15th Asia Joint Conference on Information Security (AsiaJCIS)*, pages 74–81, 2020.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[Kol18] Vladimir Kolesnikov. Free IF: How to omit inactive branches and implement S-universal garbled circuit (almost) for free. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 34–58. Springer, Heidelberg, December 2018.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, December 2001.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[se19] swisspost evoting. E-voting system 2019. `https://gitlab.com/swisspost-evoting/e-voting-system-2019`, 2019.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[Zav20] Greg Zaverucha. The picnic signature algorithm. Technical report, 2020. `https://github.com/microsoft/Picnic/raw/master/spec/spec-v3.0.pdf`.

A Blum87 is Stackable: Proof of Lemma 2

Let $\mathcal{L}_n^{\text{Ham}} \subseteq \{0,1\}^{n\times n}$ be the language of n vertex graphs with a Hamiltonian cycle (represented by adjacency matrices). For any n Blum's classical Σ-protocol for $\mathcal{L}_n^{\text{Ham}}$ is stackable, recall the protocol (Shown in Figure 10):

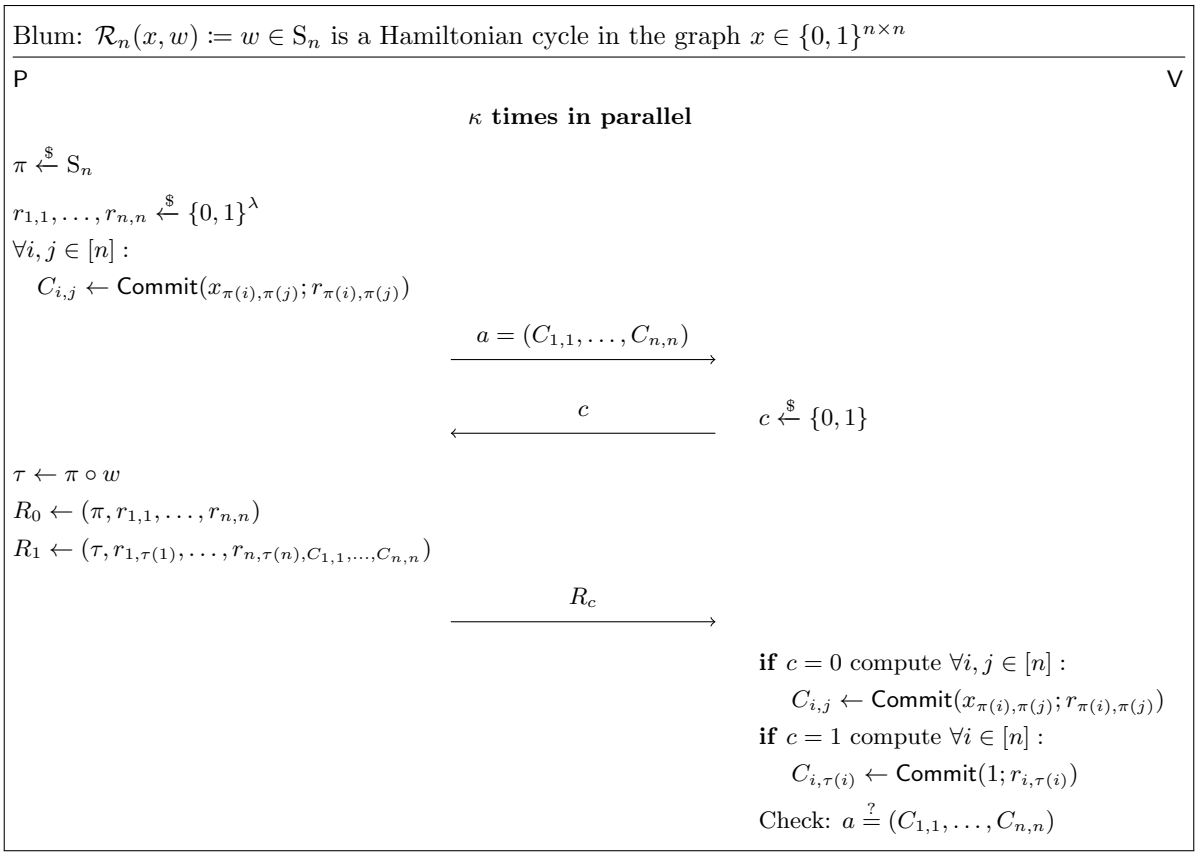

Figure 10: Blum's protocol for Hamiltonian cycles. S_n denotes the permutation group on $[n]$ and $\circ$ is the group operation. $\{0,1\}^{n\times n}$ denotes the set of adjacency matrices for n vertex graphs.

On challenge $\mathbf{c=0}$: P sends the randomness for the commitment to the permuted graph. The verifier then recomputes the commitments and checks them against the first round message. Hence for $c=0$ the last round message consists of a uniformly random permutation $\pi \in \mathrm{S}_n$ and $(\{0,1\}^\lambda)^{n^2}$ random bits, independent of the graph (statement).

On challenge $\mathbf{c=1}$: P sends the opening of the permuted Hamiltonian cycle (witness) to V. Hence for $c=1$ the last round message z is a uniformly random permutation $\tau \in \mathrm{S}_n$ and $(\{0,1\}^\lambda)^n$ random bits, independent of the graph (statement).

Therefore the protocol has recyclable third messages. To be more precise:

Proof. (Lemma 2) For $c \in \{0,1\}$, define $\mathcal{D}_c^{(z)}$ as follows:

$$\mathcal{D}_0^{(z)} := \{(\pi, r_{1,1},\ldots,r_{n,n}) \mid \pi \xleftarrow{\$} \mathrm{S}_n; \forall i,j \in [n] : r_{i,j} \xleftarrow{\$} \{0,1\}^\lambda\}$$

$$\mathcal{D}_1^{(z)} := \{(\tau, r_{1,\tau(1)}, \ldots, r_{n,\tau(n)}, C_{1,1}, \ldots, C_{n,n}) \mid \tau \xleftarrow{\$} \mathrm{S}_n; \forall i,j \in [n] : r_{i,j} \xleftarrow{\$} \{0,1\}^\lambda, C_{i,j} \leftarrow \mathsf{Commit}(1; r_{i,j})\}$$

Construct the extended simulator $\mathcal{S}^{\text{EHVZK}}$ as follows:

$$\mathcal{S}^{\text{EHVZK}}(x, c = 0, (\pi, r_{1,1}, \ldots, r_{n,n})) = (C_{1,1}, \ldots, C_{n,n}) \text{ where } \forall i,j \in [n] : C_{i,j} \leftarrow \mathsf{Commit}(x_{i,j}; r_{i,j})$$

$$\mathcal{S}^{\text{EHVZK}}(x, c = 1, (\tau, r_{1,\tau(1)}, \ldots, r_{n,\tau(n)}, C_{1,1}, \ldots, C_{n,n})) = (C_{1,1}, \ldots, C_{n,n})$$

Observe that the distribution for $c = 0$ is the same as honest execution. For $c = 1$ the distributions are indistinguishable by hiding of the bit-commitment, see [Blu87] for details. The protocol is clearly EHVZK – since it is a special case of a commit-and-reveal protocol. □

B Well-Behaved Simulators: Proof of Lemma 5

Proof. Given $\Pi = (A, Z, \phi)$, construct the new $\Pi' = (A', Z', \phi')$ with well-behaved a simulator as follows:

- $A'(x, w; r) \coloneqq (a, \perp)$ where $a \leftarrow A(x, w; r)$
- $Z'(x, w, c; r) \coloneqq z$ where $z \leftarrow Z(x, w, c; r)$
- $\phi'(x, a', c, z) \coloneqq$
  1. **if** $a' = (\perp, c)$ output 1.
  2. **if** $a' = (a, \perp)$ for some a, output $\phi(x, a, c, z)$.
  3. Otherwise output 0

Intuitively: in Π' the prover can either choose to attempt guessing the challenge c (sending $a' = (\perp, c)$), or, he can run the original protocol (sending $a' = (a, \perp)$). The (well-behaved) simulator $\mathcal{S}'$ of Π' first runs the simulator $\mathcal{S}$ of Π, if the simulated transcript (a, c, z) is accepting then output the transcript, otherwise $\mathcal{S}'$ 'guesses' the challenge:

- $\mathcal{S}(1^\lambda, x, c) \coloneqq$
  1. $(a, z) \leftarrow \mathcal{S}(1^\lambda, x, c)$
  2. **if** $\phi(a, c, z) = 1$ output (a', z') where $a' = (a, \perp)$, $z' = z$.
  3. **if** $\phi(a, c, z) = 0$ output (a', z') where $a' = (\perp, c)$, $z' = \perp$

Π' is a Σ-protocol: Formally verify the defining qualities of a Σ-protocol:

- Completeness: follows from completeness of Π. In particular in the real executions $a' = (a, \perp)$ always.
- Special Honest Verifier Zero-Knowledge: For every $x \in \mathcal{L}$ and $c \in \{0,1\}^\lambda$, the output of the original simulator $(a, z) \leftarrow \mathcal{S}(1^\lambda, x, c)$ must always be accepting $\phi(a, c, z) = 1$ by SHVZK of Π. Hence the distribution of $\mathcal{S}'$ on statements $x \in \mathcal{L}$ is Since the distribution in the real execution will always have $a' = (a, \perp)$
- Special Soundness: Suppose we get two transcripts with a shared first-round message: (a', c_1, z_1, c_2, z_2) st. $\phi'(a', c_1, z_1) = 1$, $\phi(a', c_2, z_2) = 1$ and $c_1' = c_2'$. Consider the two distinct forms that a' can take:
  1. When $a' = (\perp, c)$ then clearly there does not exists two accepting transcripts with different challenges c_1 and c_2 since $c = c_1 = c_2$. Hence the assumption that $a' = (\perp, c)$ is a contradiction.
  2. When $a' = (a, \perp)$ then z_1, z_2 must satisy $\phi(a, c_1, z_1) = 1$ and $\phi(a, c_2, z_2) = 1$. Therefore, we can extract a witness $w \leftarrow \mathcal{E}(a, c_1, z_1, c_2, z_2)$ using the extractor of Π.

Π is EHVZK $\implies$ Π' is EHVZK: Let $\mathcal{S}^{\text{EHVZK}}$ be the extended simulator of Π, for every x define $\mathcal{D}_{c,x}^{(z)'} = \mathcal{D}_{c,x}^{(z)}$ and the new extended simulator $\mathcal{S}^{\text{EHVZK}'}$ of Π' as:

- $\mathcal{S}^{\text{EHVZK}'}(1^\lambda, x, c, z) \coloneqq$
  1. $a \leftarrow \mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z)$
  2. **if** $\phi(x, a, c, z) = 1$: output $(a, \bot)$
  3. **if** $\phi(x, a, c, z) = 0$: output $(\bot, c)$

Π has recyclable third messages $\implies$ Π' has recyclable third messages: Let $\mathcal{D}_c^{(z)'} = \mathcal{D}_c^{(z)}$, since $\mathcal{D}_{c,x}^{(z)'} = \mathcal{D}_{c,x}^{(z)}$ for every x, it follows immediately. □

C Security Proof for Cross-Stacking Compiler (Theorem 6)

We now prove that the protocol $\Pi' = (A', Z', \phi')$ described in Figure 7 is a stackable Σ-protocol for the relation $\mathcal{R}'((x_1, \ldots, x_\ell), (\alpha, w)) \coloneqq \mathcal{R}_i(x_\alpha, w)$.

Completeness. Completeness follows directly from the completeness of the underlying Σ-protocols, completeness of the commitment scheme. Note that because the underlying Σ-protocol has a well-behaved simulator, the prover will not produce non-accepting transcripts on any clauses embedding false instances.

Special Soundness. We create an extractor $\mathcal{E}'$ for the protocol Π' using the extractors $\mathcal{E}_i$ for the underlying Σ-protocols Π_i. The extractor $\mathcal{E}'$ is given two accepting transcripts for the protocol Π' that share a first round message, *i.e.* a, c, z, c', z'. The extractor uses this input to recover 2ℓ total transcripts (2 for each branch), $(a_i, c, z_i), (a_i', c', z_i')$ for $i \in [\ell]$. By the binding and verification properties of the equivocal vector commitment scheme, with all but negligible probability there exists an $\alpha \in [\ell]$ such that $a_\alpha = a_\alpha'$. $\mathcal{E}'$ then invokes the extractor of Π_α on these transcripts to recover $w \leftarrow \mathcal{E}_\alpha(1^\lambda, x_\alpha, a_\alpha, c_\alpha, z_\alpha, c_\alpha', z_\alpha')$ and returns (α, w). Because the underlying extractor $\mathcal{E}_\alpha$ cannot fail with non-negligible probability, the $\mathcal{E}'$ succeeds with overwhelming probability.

Extended Honest-Verifier Zero-Knowledge (and Recyclable Third Messages). We denote the distribution of third round message for Π' as $\mathcal{D}_c^{(z)'}$. Note that $\mathcal{D}_c^{(z)'}$ is constructed from a commitment key ck, a randomness for the commitment scheme r, and a single element $d \in \mathcal{D}$. Note that by the hiding property of the commitment scheme, the distribution of ck is independent of the binding index B. More formally, for $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$,

$$\mathcal{D}_c^{(z)'} \coloneqq \{(\mathsf{ck}, r, d) \mid (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{1\}); r \xleftarrow{\$} \{0,1\}^\lambda; d \xleftarrow{\$} \mathcal{D}\}$$

Note that this distribution is independent of the statements, as $\mathcal{D}$ itself is independent of the statements.

We construct the extended simulator by running the underlying extended simulating $\mathcal{S}^{\text{EHVZK}}$ for every clause and committing to the tuple of first round message $(a_1, \ldots, a_\ell)$ using a freshly generated commitment key ck and randomness r:

$a' \leftarrow \mathcal{S}^{\text{EHVZK}'}((x_1, \ldots, x_\ell), c, z' = (\mathsf{ck}, r, d))$

1 : **for** $i \in [\ell]$
2 : Compute $z_i \leftarrow \mathsf{TExt}_i(c, d)$
3 : Compute $a_i \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_i)$
4 : **return** $(\mathsf{ck}, \mathsf{BindCom}(\mathsf{ck}, \mathbf{v} = (a_1, \ldots, a_\ell); r))$

Let $\mathcal{D}^{(\alpha,w)}$ denote the distribution of transcripts resulting from an honest prover possessing witness (α, w) running Π' with an honest verifier on the statement $(x_1, \ldots, x_\ell)$, where $\mathcal{D}^{(\alpha,w)}$ is over the randomness of the

prover and the verifier. We now proceed using a hybrid argument. Let $\mathcal{H}^{(\alpha)}$ be the same as $\mathcal{D}^{(\alpha,w)}$, except let the first round message of clause α be generated by simulation, *i.e.* $z_\alpha \leftarrow \mathsf{TExt}_\alpha(c,d); a_\alpha \leftarrow \mathcal{S}^{\text{EHVZK}}(x_\alpha, c, z_\alpha)$. By the EHVZK of Σ_α, $\mathcal{H}^{(\alpha)} \approx \mathcal{D}^{(\alpha,w)}$. Next, let $\mathcal{H}^{(\alpha,\mathsf{ck})}$ be the same as $\mathcal{H}^{(\alpha)}$ except let the commitment key ck be generated with the binding position as $B = \{1\}$, *i.e.* $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \{1\})$. Observe that $\mathcal{H}^{(\alpha,\mathsf{ck})} \stackrel{p}{=} \mathcal{H}^{(\alpha)}$ by the (perfect) hiding of the partially-binding commitment scheme. Lastly note that $\mathcal{H}^{(\alpha,\mathsf{ck})}$ matches the output distribution of $\mathcal{S}^{\text{EHVZK}\prime}((x_1,\ldots,x_\ell), c, \mathcal{D}_c^{(z)\prime})$.

Therefore Σ' is a stackable Σ-protocol.

D Overview of [KKW18] and Proof of Lemma 3

In this section we describe the MPC-in-the-head protocol by Katz, Kolesnikov and Wang (KKW) [KKW18]. Let R be a finite commutative ring and $m \in \mathbb{N}_+$, KKW [KKW18] (parameterized by R and m) is a Σ-protocol for the NP relation $\mathcal{R}(C, w) \coloneqq C(w) = 1$ of circuits C over R and satifying assignments of input wires w. The protocol is obtain by compiling a passively secure BGW-style [BGW88] MPC protocol in the preprocessing model using the IKOS [IKOS07] compiler.

Notation. We denote by $v \xleftarrow{\$}_s \mathcal{D}$ (notice s), the process of sampling v from the distribution $\mathcal{D}$ using random coins $r \leftarrow \mathsf{PRG}(s)$ derived by applying a pseudo-random generator on the seed s. The operation is stateful i.e. $v_1 \xleftarrow{\$}_s \mathcal{D}; v_2 \xleftarrow{\$}_s \mathcal{D}$ samples two (possibly distinct) values from $\mathcal{D}$ using disjoint slices of the pseudo-random stream. We denote by $[v]^{(i)}$ the additive share of v held by player P_i, the shares of all players sum to v: $v = \sum_{i=1}^{n} [v]^{(i)}$. The function f to be computed by the MPC protocol is implemented as an arithmetic circuit over R and the function is interpreted as a sequence of gates $f = (f_1, \ldots, f_{|f|}) \in \{\texttt{Input}, \texttt{Add}, \texttt{Mul}, \texttt{Output}\}^{|f|}$ (in-order traversal of the circuit) where:

- $\texttt{Input}(\gamma)$: assigns to the next R-element from the witness to the wire w_γ.
- $\texttt{Add}(\gamma, \alpha, \beta)$: assigns to the wire w_γ the sum of the values of the wires w_α and w_β.
- $\texttt{Mul}(\gamma, \alpha, \beta)$: assigns to the wire w_γ the product of the values of the wires w_α and w_β.
- $\texttt{Output}(\alpha)$: outputs/reveals the value of the wire w_α.

Underlying MPC. It is most clearly seen how preprocessing as used in KKW fits into our framework by simply viewing the MPC as an $n+1$ player protocol, where a 'preprocessing player' P_0 acts as a dealer (shown in Figure 11) and sends the 'online players' $P_1, \ldots, P_n$ correlated randomness via point-to-point channels. The MPC is passively secure against the corruption patterns $\mathcal{C} = \{\{0\} \cup \binom{[n]}{n-1}\}$ i.e. the 'preprocessing player' or any $n-1$ subset of the 'online players'. During the online phase of KKW (shown in Figure 12) the n players hold additive shares $[\lambda_\gamma]^{(i)}$ of masks $\lambda_\gamma = \sum_{i=1}^{n} [\lambda_\gamma]^{(i)}$ and public maskings $z_\gamma = v_\gamma - \lambda_\gamma$ of the value v_γ assigned to the w_γ, i.e. the value of w_γ is $v_\gamma = z_\gamma + \sum_{i=1}^{n} [\lambda_\gamma]^{(i)}$.

The n online players share a single broadcast channel (no point-to-point channels). The initial state of the online players consists of the masked values z_γ for the input gates sent to the players on the broadcast channel. The initial state of the preprocessing player P_0 consists only of random coins. Player 0 can be opened by providing her random coins, any subset of the n online players can be opened by providing the messages from player 0 to these players in addition to the messages broadcast during the online execution by the unopened players.

When applying the IKOS [IKOS07] compiler to this $n+1$ player MPC protocol, it results in the 3-round (Σ-protocol) variant of the KKW proof system described in the original paper. The communication-complexity optimizations applied in [KKW18] are compatible with this $n+1$ player interpretation, but are omitted here for the sake of simplicity and because they are orthogonal to our goal of 'stacking' KKW.

Player 0 computes correlated Beaver triples:

- Sample a PRG seed for every player **for** $i \in [n] : s_i \xleftarrow{\$}_{s_0} \{0,1\}^\lambda$.
- Create an empty list of 'corrected' multiplication shares: $\Delta \leftarrow \emptyset$
- Process the circuit gate-by-gate **for** $j \in [|f|]$ do:
  - **if** $f_j = \texttt{Input}(\gamma)$:
    1. Sample a random sharing: **for** $i \in [n] : [\lambda_\gamma]^{(i)} \xleftarrow{\$}_{s_i} R$
  - **if** $f_j = \texttt{Add}(\gamma, \alpha, \beta)$:
    1. Locally add shares: $[\lambda_\gamma] \leftarrow [\lambda_\alpha] + [\lambda_\beta]$
  - **if** $f_j = \texttt{Mul}(\gamma, \alpha, \beta)$:
    1. Compute the product of the masks: $\lambda_{\alpha,\beta} \leftarrow \lambda_\alpha \cdot \lambda_\beta$
    2. Sample random output mask: **for** $i \in [n] : [\lambda_\gamma]^{(i)} \xleftarrow{\$}_{s_i} R$
    3. Sample random shares of the product: **for** $i \in [n] : [\lambda_{\alpha,\beta}]^{(i)} \xleftarrow{\$}_{s_i} R$
    4. Compute the correction $\Delta_{\alpha,\beta} \leftarrow \lambda_{\alpha,\beta} - \sum_{i=1}^{n} [\lambda_{\alpha,\beta}]^{(i)}$
    5. Append $\Delta_{\alpha,\beta}$ to Δ.
- Send correlated randomness to each player: **for** $i \in [n]$ send (Δ, s_i) to P_i

Figure 11: KKW Preprocessing Player. R is any finite commutative ring.

Condensed Views

Condensed view of P_0: The condensed view of P_0 is its random coins s_0 from which the individual player seeds $s_1, \ldots, s_n$ are derived. Given s_0 the entire view of P_0 can be recomputed. Total of λ bits.

Condensed view of $\{P_i\}_{i\in\mathcal{I}}, 0 \notin \mathcal{I}$: The condensed views of any subset of online players consists of a tuple $(\mathcal{T}, \Delta, \{s_i\}_{i\in\mathcal{I}})$, consisting of:

1. All broadcast messages not sent by players in $\{P_i\}_{i\in\mathcal{I}}$:
   (a) The masked input wires z_γ for gates $\texttt{Input}(\gamma)$.
   (b) The $[s_\gamma]^{(p)}$ shares sent by player $P_p, p \notin \mathcal{I}$ during multiplication.
2. The corrections Δ sent by player 0.
3. The $n-1$ individual per-player PRG seeds $\{s_i\}_{i\in\mathcal{I}}$,

Total of $2m + |w|$ elements of R and $\lambda \cdot (n-1)$ bits. Crucially, there is no need to include the shares of the honest player for the output reconstructions:

Remark 5. *We do not need to include in $\mathcal{T}$ the shares of P_p during the reconstruction in the execution of* `Output` *gates: any accepting transcript will reconstruct the constant o ('circuit satisfied'), hence the share $[\lambda_\alpha]^{(p)}$ can be inferred from the masked wire z_α and the shares $\{[\lambda_\alpha]^{(i)}\}_{i\in\mathcal{I}}$ as: $[\lambda_\alpha]^{(p)} = o - z_\alpha - \sum_{i\in\mathcal{I}} [\lambda_\alpha]^{(i)}$.*

Soundness Amplification. In KKW, communication complexity of the soundness amplification is improved by opening the preprocessing player with significantly higher probability. In practice this is done by picking parameters M, τ with $\tau \ll M$ then opening P_0 in $M - \tau$ randomly chosen repetitions and a random subset of 'online players' in the remaining τ repetitions.

For every wire (with secret-shared value v_γ) the players hold a public masked value $z_\gamma = v_\gamma - \lambda_\gamma$. For the input gates the masked values $z_\gamma = w_\gamma - \lambda_\gamma$ are provided to n online players on the broadcast channel before execution begins.

Player P_i with $i \in [n]$:

- Receive corrections and PRG seed (Δ, s_i) from P_0
- Process the circuit f in-order gate-by-gate **for** $j \in [|f|]$ do:
  - **if** $f_j = \texttt{Input}(\gamma)$:
    1. Receive the masked input z_γ on the broadcast channel.
    2. Regenerate the random sharing $[\lambda_\gamma]^{(i)} \xleftarrow{\$}_{s_i} \mathsf{R}$
       (players obtain a sharing of the witness $w_\gamma = z_\gamma + \sum_{i\in[n]} [\lambda_\gamma]^{(i)}$)
  - **if** $f_j = \texttt{Add}(\gamma, \alpha, \beta)$:
    1. Locally compute $[\lambda_\gamma]^{(i)} \leftarrow [\lambda_\alpha]^{(i)} + [\lambda_\beta]^{(i)}$
    2. Locally compute $z_\gamma \leftarrow z_\alpha + z_\beta$
  - **if** $f_j = \texttt{Mul}(\gamma, \alpha, \beta)$:
    1. Regenerate next output mask: $[\lambda_\gamma]^{(i)} \xleftarrow{\$}_{s_i} \mathsf{R}$
    2. Regenerate next Beaver share: $[\lambda_{\alpha,\beta}]^{(i)} \xleftarrow{\$}_{s_i} \mathsf{R}$
    3. Correct share: **if** $i = 1$ update $[\lambda_{\alpha,\beta}]^{(i)} \leftarrow [\lambda_{\alpha,\beta}]^{(i)} + \Delta_{\alpha,\beta}$
    4. Locally compute $[s_\gamma]^{(i)} \leftarrow z_\alpha[\lambda_\beta]^{(i)} + z_\beta[\lambda_\alpha]^{(i)} + [\lambda_{\alpha,\beta}]^{(i)} - [\lambda_\gamma]^{(i)}$
    5. Reconstruct s_γ (broadcast $[s_\gamma]^{(i)}$)
    6. Locally compute $z_\gamma \leftarrow s_\gamma + z_\alpha z_\beta$
  - **if** $f_j = \texttt{Output}(\alpha)$:
    1. Reconstruct λ_α (broadcast $[\lambda_\alpha]^{(i)}$)

Figure 12: KKW Online Players. R is any finite commutative ring.

D.1 [KKW18] is Stackable: Proof of Lemma 3

We now prove that this MPC is $\mathcal{F}$-universally simulatable (and therefore stackable).

Proof. (Lemma 3) Let $\mathcal{D}^{(\texttt{real})}$ be the real distribution over condensed views for a particular $\mathcal{I}$. The simulator is given in Figure 13. Consider the two cases:

- **Preprocessing:** $\mathcal{I} = \{0\}$.
  The distribution $\mathcal{S}(f, \{0\})$ over condensed views is exactly $\mathcal{D}^{(\texttt{real})}$.

- **Online Execution:** $\mathcal{I} \neq \{0\}, |\mathcal{I}| = n - 1$.
  Follows in a straighforward way from the pseudorandomness of the PRG. Consider the following three hybrids:

  1. Define the hybrid $\mathcal{H}^{(\Delta)}$:

  $$\mathcal{H}^{(\Delta)} = \{(\mathcal{T}, \Delta', I, \{s_i\}_{i\in\mathcal{I}}), \Delta' \xleftarrow{\$} R^m, (\mathcal{T}, \Delta, I, \{s_i\}_{i\in\mathcal{I}}) \xleftarrow{\$} \mathcal{D}^{(\texttt{real})}(\mathcal{I})\}$$

  Let $p \in [n] \setminus \mathcal{I}$ be the honest (unopened) player. Note that in $\mathcal{D}^{(\texttt{real})}$: $\Delta_{\alpha,\beta} = \lambda_{\alpha_j}\lambda_{\beta_j} - \sum_{i\in[i]} [\lambda_{\alpha_j,\beta_j}]^{(i)} = C_{\alpha_j,\beta_j} - [\lambda_{\alpha_j,\beta_j}]^{(p)}$ where C_{α_j,β_j} and $[\lambda_{\alpha_j,\beta_j}]^{(p)} \xleftarrow{\$}_{s_p} R$ is known to the verifier. In $\mathcal{H}^{(\Delta)}$: $\Delta'_{\alpha,\beta} = \lambda_{\alpha_j}\lambda_{\beta_j} - \sum_{i\in[i]} [\lambda_{\alpha_j,\beta_j}]^{(i)} = C_{\alpha_j,\beta_j} - [\lambda_{\alpha_j,\beta_j}]^{(p)}$ where $[\lambda_{\alpha_j,\beta_j}]^{(p)} \xleftarrow{\$} R$. Hence by pseudorandomness of the PRG the distribution of $\Delta_{\alpha,\beta}$ and $\Delta'_{\alpha,\beta}$ are computationally indistinguishable and by extension $\mathcal{D}^{(\texttt{real})} \overset{\text{c}}{\approx} \mathcal{H}^{(\Delta)}$.

  2. Define the hybrid $\mathcal{H}^{(\mathcal{T})}$:

  $$\mathcal{H}^{(\mathcal{T})} = \{(\mathcal{T}', \Delta, I, \{s_i\}_{i\in\mathcal{I}}), \mathcal{T}' \xleftarrow{\$} R^{m+|w|}, (\mathcal{T}, \Delta, I, \{s_i\}_{i\in\mathcal{I}}) \xleftarrow{\$} \mathcal{H}^{(\mathcal{T})}\}$$

  Note that in $\mathcal{D}^{(\texttt{real})}$: $[s_\gamma]^{(p)} = z_\alpha[\lambda_\beta]^{(p)} + z_\beta[\lambda_\alpha]^{(p)} + [\lambda_{\alpha,\beta}]^{(p)} - [\lambda_\gamma]^{(p)} = S^{(p)}_{\alpha,\beta} - [\lambda_\gamma]^{(p)}$ with $[\lambda_\gamma]^{(p)} \xleftarrow{\$}_{s_p} R$ and the verifier may know $S^{(p)}_{\alpha,\beta}$. While in $\mathcal{H}^{(\mathcal{T})}$: $[s_\gamma]^{(p)'} \xleftarrow{\$} R$. By pseudorandomness of the PRG the distribution of $[s_\gamma]^{(p)}$ and $[s_\gamma]^{(p)}$ are computationally indistinguishable and by extension $\mathcal{D}^{(\texttt{real})} \overset{\text{c}}{\approx} \mathcal{H}^{(\mathcal{T})}$.

  Finally observe $\mathcal{H}^{(\Delta,\mathcal{T})} = \mathcal{S}(f, \mathcal{I}) \overset{\text{c}}{\approx} \mathcal{D}^{(\texttt{real})}$.

□

E Overview of Ligero and Proof of Lemma 4

In this section, we discuss the MPC model used in Ligero, give an overview about why their underlying MPC is $\mathcal{F}$-universally simulatable , recall the construction of their MPC protocol and finally give a formal proof for why their protocol is $\mathcal{F}$-universally simulatable .

MPC Model. The protocol in Ligero [AHIV17] makes use of a special MPC protocol that is described in the following model between a sender, reciever and n servers (the following text is taken verbatim from Ligero):

- **Two-phase:** The protocol they consider proceeds in two-phases: In phase 1, the servers receive inputs from the sender and only perform local computation. After Phase 1, the servers obtain a public random string r sampled via a coin flipping oracle and broadcast to all servers. The servers use this in Phase 2 for their local computation at the end of which each server sends a single output message to the receiver R.

KKW $\mathcal{S}(f, \mathcal{I} = \{0\})$, preprocessing is opened.

Sample PRG seed: $s_0 \xleftarrow{\$} \{0,1\}^\lambda$

return s_0

KKW $\mathcal{S}(f, \mathcal{I} \neq \{0\})$, online-phase is partially opened.

Sample condensed broadcast transcript: $\mathcal{T} \xleftarrow{\$} R^{m+|w|}$

Sample per-player PRG seeds: $\forall i \in \mathcal{I} : s_i \xleftarrow{\$} \{0,1\}^\lambda$

Sample corrections: $\Delta \xleftarrow{\$} R^m$

return $(\mathcal{T}, \Delta, \{s_i\}_{i \in \mathcal{I}})$

Figure 13: Simulating the condensed views in KKW. R is the commutative ring over which the arithmetic circuits are computed.

- **No Broadcast:** The servers never communicate with each other. Each server simply receives inputs from the sender at the beginning of Phase 1, then receives a public random string in Phase 2, and finally delivers a message to R.

Overview. Originally, Ligero is presented as a 5 round public coin proof that can be flattened using Fiat-Shamir. In order to use a protocol in the above model with our modified IKOS compiler (see Theorem 3), we assume that the random string r is obtained by the sender using a random oracle by providing the list of all the messages that it computes in the first phase as input. Given this slight modification, we observe that the underlying MPC protocol in Ligero is $\mathcal{F}$-universally simulatable . At a high level, the messages sent by the sender to the servers at the end of the first phase in their protocol correspond to packed secret sharings (or more generally Reed-Solomon encodings) of the intermediate wire values obtained upon evaluating the circuit on a given input. The messages sent by the servers to the receiver in the second phase correspond to packed secret sharings of vectors of 0s. Since the messages sent in the first phase are never reconstructed, our $\mathcal{F}$-universal simulator, can simply simulate these messages by sending random values to the adversarial servers on behalf of an honest sender. These messages correspond to the condensed view of the adversary. Messages sent by the honest servers to a corrupt receiver in the second round can be deterministically computed using the above condensed view and the description of the function. Hence, this protocol is $\mathcal{F}$-universally simulatable .

We now describe their protocol in detail and then present a formal description of the $\mathcal{F}$-universal simulator and the functions ExpandViews and CondenseViews. But befor that, we borrow the following definitions from [AHIV17], which will aid in the description of the protocol.

Definition 13 (Reed-Solomon Code)**.** *For positive integers n, k, finite field $\mathbb{F}$, and a vecotr $\eta = (\eta_1, \ldots, \eta_n) \in \mathbb{F}^n$ of distinct field elements, the code $\mathsf{RS}_{\mathbb{F},n,k,\eta}$ is the $[n, k, n-k+1]$ linear code over $\mathbb{F}$ that consists of all n-tuples $(p(\eta_1), \ldots, p(\eta_n))$, where p is a polynomial of degree $< k$ over $\mathbb{F}$.*

Definition 14 (Interleaved code)**.** *Let $L \subset \mathbb{F}^n$ be an $[n, k, d]$ linear code over $\mathbb{F}$. We let L^m denote the $[n, mk, d]$ (interleaved) code over $\mathbb{F}^m$ whose codewords are all $m \times n$ matrices U such that every U_i of U satisfies $U_i \in L$. For $U \in L^m$ and $j \in [n]$, we denote by $U[j]$ the j^{th} symbol (column) of U.*

Definition 15 (Encoded Message)**.** *Let $L = \mathsf{RS}_{\mathbb{F},n,k,\eta}$ be an RS code and $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ be a sequence of distinct elements of $\mathbb{F}$ for $\ell \leq k$. For $u \in L$, we define the message $\mathsf{Dec}_\zeta(u)$ to be $(p_u(\zeta_1), \ldots, p_u(\zeta_\ell))$, where p_u is the polynomial (of degree $< k$) corresponding to u. For $U \in L^m$ with rows $u^1, \ldots, u^m \in L$, we let $\mathsf{Dec}_\zeta(U)$ be the length-$m\ell$ vector $x = (x_{11}, \ldots, x_{1\ell}, \ldots, x_{m1}, \ldots, x_{m\ell})$ such that $(x_{i1,\ldots,x_{i\ell}}) = \mathsf{Dec}_\zeta(u^i)$ for $i \in [m]$. Finally, when ζ is clear from the context, we say that U encodes x if $x = \mathsf{Dec}_\zeta(U)$.*

Ligero MPC protocol. Let $C : \mathbb{F}^n \to \mathbb{F}$ be the circuit that the parties wish to compute. Let $\alpha = (\alpha 1, \ldots, \alpha_n)$ be the input vector held by the the sender S. Let m, ℓ be integers such that $m \cdot \ell > n \cdot |C|$, where $|C|$ is the number of gates in the circuit C.

In the first phase of the protocol, the sender S proceeds as follows (the following text is taken verbatim from Ligero):

- It computes $w \in \mathbb{F}^{m\ell}$, where the first $n + s$ entries of w are $(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_{|C|})$ where β_i is the output of the i^{th} gate when evaluating $C(\alpha)$.
- It then constructs vectors x, y and z in $\mathbb{F}^{m\ell}$ where the j^{th} entry of x, y and z contains the values β_a, β_b and β_c corresponding to the j^{th} multiplication gate in w.
- It constructs matrices P_x, P_y and P_z in $\mathbb{F}^{m\ell \times m\ell}$ such that

$$x = P_x w, \quad y = P_y w, \quad z = P_z w.$$

- It then constructs matrix $P_{\mathsf{add}} \in \mathbb{F}^{m\ell \times m\ell}$ such that the j^{th} row of $P_{\mathsf{add}} w$ equals $\beta_a + \beta_b - \beta_c$ where β_a, β_b and β_c correspond to the j^{th} addition gate in w.
- It then samples random codewords $U^w, U^x, U^y, U^z \in L^m$ where $L = \mathsf{RS}_{\mathbb{F},n,k,\eta}$ subject to $w = \mathsf{Dec}_\zeta(U^w), x = \mathsf{Dec}_\zeta(U^x), y = \mathsf{Dec}_\zeta(U^y), z = \mathsf{Dec}_\zeta(U^z)$ where $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ is a sequence of distinct elements disjoint from $(\eta_1, \ldots, \eta_n)$.
- Let $u', u^x, u^y, u^z, u^0, u^{\mathsf{add}}$ be auxiliary rows sampled randomly from L where each of $u^x, u^y, u^z, u^{\mathsf{add}}$ encodes an independently samples random ℓ messages $(\gamma_1, \ldots, \gamma_\ell)$ subject to $\Sigma_{c\in[\ell]}\gamma_c = 0$ and u^0 encodes 0^ℓ.
- It sets $U \in L^{4m}$ as a juxtaposition of the matrices $U^w, U^x, U^y, U^z \in L^m$. It also computes $r^* \leftarrow H^{\mathsf{RO}}(U)$, where $r^* = (r, r^{\mathsf{add}}, r^x, r^y, r^z, r^q)$, such that $r \in \mathbb{F}^{4m}, r^{\mathsf{add}}, r^x, r^y, r^z \in \mathbb{F}^{m\ell}$, $r^q \in \mathbb{F}^m$.
- It sends $U[j], u'[j], u^x[j], u^y[j], u^z[j], u^0[j], u^{\mathsf{add}}[j]$ to server j (for $j \in [n]$), where $U[j]$ represents the j^{th} column in U. It also sends r^* to each server.

In the second phase, each server $j \in [n]$ computes and broadcast the following to the receiver party R:

- Compute and send $v[j] = r^T U[j] + u'[j]$.
- – Compute constructs matrix $P_{\mathsf{add}} \in \mathbb{F}^{m\ell \times m\ell}$ such that the j^{th} row of $P_{\mathsf{add}} w$ equals $\beta_a + \beta_b - \beta_c$ where β_a, β_b and β_c correspond to the j^{th} addition gate in w.[19]
  - Let r_i^{add} be the unique polynomial of degree $< \ell$ such that $r_i^{\mathsf{add}}(\zeta_c) = ((r^{\mathsf{add}})^T P_{\mathsf{add}})_{ic}$ for every $c \in [\ell]$.
  - Let $U^w[i, j]$ be the $(i, j)^{\text{th}}$ entry in U^w.
  - Compute and send $q^{\mathsf{add}}[j] = u^{\mathsf{add}}[j] + \Sigma_{i\in[m]} r_i^{\mathsf{add}}(j) \cdot U^w[i, j]$.
- It constructs matrices P_x, P_y and P_z in $\mathbb{F}^{m\ell \times m\ell}$ such that $x = P_x w, y = P_y w, z = P_z w$.. For each $a \in \{x, y, z\}$, let r_i^a be the unique polynomial of degree $< \ell$ such that $r_i^a(\zeta_c) = ((r^a)^T [I_{m\ell}| - P_a])_{ic}$ for every $c \in [\ell]$. It then computes and sends the following:
  - $q^x[j] = u^x[j] + \Sigma_{i\in[m]} r_i^x(j) \cdot U^x[i, j] + \Sigma_{i=m+1}^{2m} r_i^x(j) \cdot U^w[i - m, j]$.
  - $q^y[j] = u^y[j] + \Sigma_{i\in[m]} r_i^y(j) \cdot U^y[i, j] + \Sigma_{i=m+1}^{2m} r_i^y(j) \cdot U^w[i - m, j]$.
  - $q^z[j] = u^z[j] + \Sigma_{i\in[m]} r_i^z(j) \cdot U^z[i, j] + \Sigma_{i=m+1}^{2m} r_i^z(j) \cdot U^w[i - m, j]$.
  - $p_0[j] = u^0[j] + \Sigma_{i\in[m]} r^q[i] \cdot (U^x[i, j] \cdot U^y[i, j] - U^z[i, j])$.

[19] Note that P_{add} can be constructed without knowledge of w.

E.1 Ligero is Stackable: Proof of Lemma 4

We now prove that the Liegor MPC is $\mathcal{F}$-universally simulatable (and therefore stackable). Based on Ligero's MPC model, privacy only holds when the adversary is only allowed to corrupt the receiver R and at most t servers. The view of an adversary corrupting the reciever R and t servers consists of the messages received by the corrupt servers from the sender S in the first phase and in the second phase it consists of the messages sent by all the servers to the receiver R. $\mathcal{F}$-universal simulatability of this protocol follows from the zero-knowledge property of Ligero. The $\mathcal{F}$-universal simulator would proceed as follows:

- Sample a random vector $v \in \mathbb{F}$.
- For each $j \in \mathcal{I}$, sample random elements from $\mathbb{F}$ for $U^x[j], U^y[j], U^z[j], U^w[j]$.
- For each $j \in \mathcal{I}$, sample random elements from $\mathbb{F}$ for $u'[j], u^x[j], u^y[j], u^z[j], u^0[j], u^{\mathsf{add}}[j]$.

Since the messages computed by the simulator are independent of the functionality (or even the output of the protocol), it is easy to see that this is an $\mathcal{F}$-universal simulator.
ExpandViews : We now describe the expand views function for this protocol

- For each $j \in \mathcal{I}$, compute the following:
  - $q^{\mathsf{add}}[j] = u^{\mathsf{add}}[j] + \Sigma_{i\in[m]} r_i^{\mathsf{add}}(j) \cdot U^w[i,j]$.
  - $q^x[j] = u^x[j] + \Sigma_{i\in[m]} r_i^x(j) \cdot U^x[i,j] + \Sigma_{i=m+1}^{2m} r_i^x(j) \cdot U^w[i-m,j]$.
  - $q^y[j] = u^y[j] + \Sigma_{i\in[m]} r_i^y(j) \cdot U^y[i,j] + \Sigma_{i=m+1}^{2m} r_i^y(j) \cdot U^w[i-m,j]$.
  - $q^z[j] = u^z[j] + \Sigma_{i\in[m]} r_i^z(j) \cdot U^z[i,j] + \Sigma_{i=m+1}^{2m} r_i^z(j) \cdot U^w[i-m,j]$.
  - $p_0[j] = u^0[j] + \Sigma_{i\in[m]} r^q[i] \cdot (U^x[i,j] \cdot U^y[i,j] - U^z[i,j])$.
- Use $\{q^{\mathsf{add}}[j]\}_{j\in\mathcal{I}}$ to extrapolate a polynomial q^{add} of degree $< k+\ell-1$ such that $\Sigma_{c\in[\ell]} q^{\mathsf{add}}(\zeta_c) = 0$, and output $\{q^{\mathsf{add}}[j]\}_{j\in[n]\setminus\mathcal{I}}$
- For each $a \in \{x,y,z\}$, use $\{q^a[j]\}_{j\in\mathcal{I}}$ to extrapolate a polynomial q^a of degree $< k+\ell-1$ such that $\Sigma_{c\in[\ell]} q^a(\zeta_c) = 0$ and output $\{q^a[j]\}_{j\in[n]\setminus\mathcal{I}}$.
- Use $\{q^0[j]\}_{j\in\mathcal{I}}$ to extrapolate a polynomial q^0 of degree $< 2k-1$ such that $p_0(\zeta_c) = 0$ for every $c \in [\ell]$ and output $\{q^0[j]\}_{j\in[n]\setminus\mathcal{I}}$.

CondenseViews: We now describe the condense views function for this protocol. This function simply removes $\{q^{\mathsf{add}}[j]\}_{j\in[n]\setminus\mathcal{I}}$, $\{q^0[j]\}_{j\in[n]\setminus\mathcal{I}}$ and $\{q^a[j]\}_{j\in[n]\setminus\mathcal{I}}$ for each $a \in \{x,y,z\}$ from the views and outputs the remaining transcript as the condensed views.

F k-out-of-ℓ Proofs Of Partial Knowledge (Old Construction)

In this section, we show how our cross stacking compiler can be extended to obtain efficient k-out-of-ℓ proofs of partial knowledge. The communication complexity in the protocol is linear in both k and ℓ. We start by formally defining k-out-of-ℓ binding vector-of-vector commitments.

Definition 16 (k-out-of-ℓ Binding Vector-of-Vector Commitment)**.** *A k-out-of-ℓ binding non-interactive vector commitment scheme with message space $\mathcal{M}$, is defined by a tuple of the PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv}, \mathsf{Open})$ *defined as follows:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ *On input the security parameter λ, the setup algorithm outputs public parameters* pp.
- $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$*: Takes public parameters* pp *and a k-subset of indices $B \in \binom{[\ell]}{k}$. Returns a commitment key* ck *and equivocation key* ek.

- $(\mathsf{com}, \mathsf{op}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}^{(i)}\}_{i\in[k]}; r)$*: Takes public parameter* pp*, equivocation key* ek*,* t *ℓ-tuples* $\{\mathbf{v}^{(i)}\}_{i\in[t]}$ *and randomness* r*. Returns a partially-binding commitment* com *as well as some auxiliary equivocation information* aux*.*

- $\mathsf{op} \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}^{(i)}\}_{i\in[t]}, \{\mathbf{v}'^{(i)}\}_{i\in[k]}, \mathsf{aux})$*: Takes public parameters* pp*, equivocation key* ek*, original commitment vectors* $\{\mathbf{v}^{(i)}\}_{i\in[k]}$ *and updated commitment vectors* $\{\mathbf{v}'^{(i)}\}_{i\in[k]}$ *such that if* b_i *is the* i^{th} *element in* $B : v^{(i)}_{b_i} = v'^{(i)}_{b_i}$*, and auxiliary equivocation information* aux*. Returns equivocation randomness* r*. (Note the difference in notation from Section 5).*

- $\{0,1\} \leftarrow \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}^{(i)}\}_{i\in[k]}, \mathsf{op})$*: Takes public parameter* pp*, commitment key* ck*, commitment* com*,* k *ℓ-tuples* $\{\mathbf{v}_i\}_{i\in[k]}$ *and randomness* r *and* $\{0,1\}$*.*

The properties satisfied by the above algorithms are as follows:

(Perfect) Hiding: *The commitment key* ck *(perfectly) hides the binding positions* B *and commitments* com *(perfectly) hides the* $k \times \ell$ *committed values in the vectors. Formally, for all sets of vectors* $\{\mathbf{v}^{(i)}\}_{i\in[k]}, \{\mathbf{w}^{(i)}\}_{i\in[k]} \in \mathcal{M}^{k\times\ell}$*,* $B, B' \in \binom{[\ell]}{k}$*, and* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*:*

$$\left[(\mathsf{ck},\mathsf{com}) \;\middle|\; \begin{array}{c}(\mathsf{ck},\mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B); r \xleftarrow{\$} \{0,1\}^\lambda;\\ (\mathsf{com},\mathsf{op},\mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp},\mathsf{ek},\{\mathbf{v}^{(i)}\}_{i\in[k]}; r)\end{array}\right] \stackrel{p}{=} \left[(\mathsf{ck}',\mathsf{com}') \;\middle|\; \begin{array}{c}(\mathsf{ck}',\mathsf{ek}') \leftarrow \mathsf{Gen}(\mathsf{pp}, B'); r' \xleftarrow{\$} \{0,1\}^\lambda\\ (\mathsf{com}',\mathsf{op}',\mathsf{aux}') \leftarrow \mathsf{EquivCom}(\mathsf{pp},\mathsf{ek}',\{\mathbf{w}^{(i)}\}_{i\in[k]}; r');\end{array}\right]$$

(Computational) Partial Binding with Structure: *It is intractable for an adversary that generates the commitment key* ck *to equivocate on more than* $\ell - 1$ *positions for any given vector in the commitment. To formalize this property, we consider the class of adversaries* $\mathcal{A}_k$ *that produces a single commitment key* ck *and* t *equivocations to a commitment under that key. We denote the* j^{th} *opening to the commitment as the set* $\{\mathbf{v}^{(i),(j)}\}_{i\in[k]}$ *and the* m^{th} *element of a vector* $\mathbf{v}^{(i),(j)}$ *vector as* $v^{(i),(j)}_m$*. For all* $t \in \mathrm{poly}(\lambda)$ *and all PPT algorithms* $\mathcal{A}_k$*, if* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ *and* $(\mathsf{ck}, \mathsf{com}, \{\mathbf{v}^{(i),(1)}\}_{i\in[k]}, \ldots, \{\mathbf{v}^{(i),(t)}\}_{i\in[k]}, \mathsf{op}_1, \ldots, \mathsf{op}_t) \leftarrow \mathcal{A}_t(1^\lambda, \mathsf{pp})$*, then*

$$\Pr\Big[\nexists S \subset [\ell], |S| \geq k, \text{ and a bijection } f : S \leftrightarrow [t] \text{ s.t. } \forall s_i \in S, v^{(s_i),(1)}_{f(s_i)} = \ldots = v^{(s_i),(t)}_{f(s_i)} \;\Big|\; \mathsf{Open}(\mathsf{pp},\mathsf{ck},\mathsf{com},\{\mathbf{v}^{(i),(1)}\}_{i\in[k]},\mathsf{op}_1) = \ldots = \mathsf{Open}(\mathsf{pp},\mathsf{ck},\mathsf{com},\{\mathbf{v}^{(i),(t)}\}_{i\in[k]},\mathsf{op}_t) = 1\Big] \leq \mathsf{negl}(\lambda)$$

Partial Equivocation: *We denote the* i^{th} *element of* B *(in order) as* b_i*. Given a commitment to* $\{\mathbf{v}^{(i)}\}_{i\in[k]}$ *under a commitment key* $\mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$*, it is possible to equivocate to any* $\{\mathbf{w}^{(i)}\}_{i\in[k]}$ *as long as for* $i \in [t], v^{(i)}_{b_i} = w^{(i)}_{b_i}$*. More formally, for all* $B \in \binom{[\ell]}{k}$*, and all* $\{\mathbf{v}^{(i)}\}_{i\in[k]}, \{\mathbf{w}^{(i)}\}_{i\in[k]} \in \mathcal{M}^{k\times\ell}$ *st.* $\forall i \in [k] : v^{(i)}_{b_i} = w^{(i)}_{b_i}$ *then:*

$$\Pr\left[\mathsf{Open}(\mathsf{pp},\mathsf{ck},\mathsf{com},\{\mathbf{w}^{(i)}\}_{i\in[t]},\mathsf{op}') = 1 \;\middle|\; \begin{array}{c}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); r \xleftarrow{\$} \{0,1\}^\lambda;\\ (\mathsf{ck},\mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B);\\ (\mathsf{com},\mathsf{op},\mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp},\mathsf{ek},\{\mathbf{v}^{(i)}\}_{i\in[k]}; r);\\ \mathsf{op}' \leftarrow \mathsf{Equiv}(\mathsf{ek},\{\mathbf{v}^{(i)}\}_{i\in[k]},\{\mathbf{w}^{(i)}\}_{i\in[k]},\mathsf{aux})\end{array}\right] = 1$$

Construction in the Random Oracle Model. Let NICom be a non-interactive commitment scheme, $\mathsf{NISetCom}$, and (P, V) be a non-interactive zero-knowledge proof system in the random oracle model. We now present a construction of a k-out-of-ℓ binding vector-of-vectors commitment in Figure 14.

k-out-of-ℓ Binding Vector-of-Vectors Commitment

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$: Output $\mathsf{pp} = \perp$.
- $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$: Output $(\mathsf{ck}, \mathsf{ek}) = \perp$
- $(\mathsf{com}, \mathsf{op}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}^{(i)}\}_{i\in[t]}; r)$:
  - Parse a random value r_B from r and compute $\mathsf{com}_B = \mathsf{NISetCom}(B; r_B)$
  - For each $i \in [k], j \in [\ell]$, parse a random value $r_{i,j}$ from r and compute $\mathsf{com}_{i,j} = \mathsf{NICom}(v_j^{(i)}; r_{i,j})$.
  - Use the non-interactive zero-knowledge proof to compute a proof π using statement $\mathsf{com}_B, \{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}, \{\mathbf{v}^{(i)}\}_{i\in[t]}$ and witness $B, r_B, \{r_{i,b_i}\}_{i\in[k]}$:

$$\pi = \mathsf{NIZKPoK}\left\{\left(\{\mathbf{v}^{(i)}\}_{i\in[t]}, B, r_B, \{r_{i,b_i}\}_{i\in[k]}\right) : B \in \binom{[\ell]}{k} \ \wedge \ \mathsf{com}_B = \mathsf{NISetCom}(B; r_b) \ \wedge \ \mathsf{com}_{i,b_i} = \mathsf{NICom}(v_j^{(i)}; r_{i,b_i})\right\}$$

  - Set $\mathsf{com} = (\mathsf{com}_B, \{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]})$, $\mathsf{op} = \pi$, and $\mathsf{aux} = r$.
- $\mathsf{op} \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}^{(i)}\}_{i\in[t]}, \{\mathbf{v}'^{(i)}\}_{i\in[t]}, \mathsf{aux})$: Use the non-interactive zero-knowledge proof to compute a proof π using statement $\mathsf{com}_B, \{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}, \{\mathbf{v}'^{(i)}\}_{i\in[t]}$ and witness $B, r_B, \{r_{i,b_i}\}_{i\in[k]}$, for the above language $\mathcal{L}$ and output $\mathsf{op} = \pi$.
- $\{0,1\} \leftarrow \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}^{(i)}\}_{i\in[t]}, \mathsf{op})$: If op verifies with respect to com and $\{\mathbf{v}^{(i)}\}_{i\in[t]}$, return 1. Otherwise return 0.

Figure 14: A k-out-of-ℓ Binding Vector-of-Vectors Commitment

Theorem 7. *Let* NICom *be a non-interactive t-out-ℓ binding vector commitment scheme and* (P, V) *be a non-interactive zero-knowledge proof system in the random oracle model, then the scheme presented in Figure 14 is a non-interactive k-out-of-ℓ binding vector-of-vectors commitment in the random oracle model.*

Proof. Hiding follows from the hiding of the underlying commitment scheme. Binding and equivocation follow from the soundness and completeness of the zero-knowledge protocol. □

We now present our compiler for proofs of partial knowledge.

Theorem 8 (Stacking for Proofs of Partial Knowledge)**.** *Let $\mathcal{D}$ be a distribution. For each $i \in [\ell]$, let $\Pi_i = (A_i, C_i, Z_i, \phi_i)$ be a stackable (See Definition 9) Σ-protocol for the NP relation $\mathcal{R}_i : \mathcal{X}_i \times \mathcal{W}_i \to \{0,1\}$, that is cross simulatable w.r.t. to a distribution $\mathcal{D}$, and let* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv}, \mathsf{Open})$ *be a k-out-of-ℓ binding vector-of-vectors commitment scheme (See Definition 16). For any $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, the protocol $\Pi' = (A', C', Z', \phi')$ described in Figure 15 is a Σ-protocol for the relation $\mathcal{R}'((x_1, \ldots, x_\ell), (\mathbb{K} = \{k_1, \ldots, k_k\}, \{w_j\}_{j\in\mathbb{K}})) := \wedge_{j\in\mathbb{K}}\mathcal{R}_j(x_j, w_j)$.*

Completeness: Completeness follows directly from the completeness of the Σ-protocols, k-out-of-ℓ binding vector of vectors commitment scheme and the EHVZK simulators.

Special Soundness: Special soundness follows from the the binding property and the verification property of the commitment scheme in a similar way as in the proof of the cross stacking compiler.

Special Honest-Verifier Zero Knowledge: Special Honest Verifier Zero-Knowledge property also follows exactly like in the proof of the cross stacking compiler.

Stacking Compiler for k-out-of-ℓ Proofs of Partial Knowledge

Statement: $x = x_1, \ldots, x_n$
Witness: $w = (\mathbb{K} = \{k_1, \ldots, k_k\}, \{w_j\}_{j \in \mathbb{K}})$

- **First Round:** Prover computes $A'(x, w; r^p) \to a$ as follows:
  - Parse $r^p = (\{r_j^p\}_{j \in \mathbb{K}} \| r \| \{r_{\text{map},j}\}_{j \in [k]})$.
  - For each $j \in \mathbb{K}$, compute $a_j \leftarrow A_j(x_j, w_j; r_j^p)$.
  - For each $j \in [k]$, set $v_{j,k_j} = a_{k_j}$.
  - For each $j \in [k], i \in [\ell] \setminus k_j$, set $v_{j,i} = 0$.
  - Compute $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B = \mathbb{K})$.
  - For each $j \in [k]$, set $\mathbf{v}_j = (v_{j,1}, \ldots, v_{j,\ell})$
  - Compute $(\mathsf{com}, \mathsf{op}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ck}, \{\mathbf{v}_j\}_{j \in [k]}; r)$.
  - Send $a = \mathsf{com}$ to the verifier.
- **Second Round:** Verifier samples $c \leftarrow \{0,1\}^\lambda$ and sends it to the prover.
- **Third Round:** Prover computes $Z'(x, w_\alpha, c; r^p) \to z$ as follows:
  - Parse $r^p = (\{r_j^p\}_{j \in \mathbb{K}} \| r \| \{r_{\text{map},j}\}_{j \in [k]})$.
  - For each $j \in [k]$:
    - Compute $z_j \leftarrow Z_{k_j}(x_{k_j}, w_{k_j}, c; r_{k_j}^p)$
    - Compute $d_j \leftarrow F_{\Pi_{k_j} \to \mathcal{D}}(z_j; r_{\text{map},j})$
    - For $i \in [\ell] \setminus k_j$,
      - Set $z_{j,i} \leftarrow \mathsf{TExt}_i(c, d_j)$
      - Set $a_{j,i} \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_{j,i})$
    - Set $a_{j,k_j} \leftarrow a_{k_j}$
  - For each $j \in [k]$, set $\mathbf{v}'_j = (a_{j,1}, \ldots, a_{j,\ell})$
  - Compute $\mathsf{op}' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathsf{com}, \{\mathbf{v}_j\}_{j \in [k]}, \{\mathbf{v}'_j\}_{j \in [k]}, \mathsf{aux})$
  - Compute and send $z = \left(\mathsf{ck}, \{d_j\}_{j \in [k]}, \mathsf{op}'\right)$ to the verifier.
- **Verification:** Verifier computes $\phi'(x, a, c, z) \to b$ as follows:
  - Parse $a = \mathsf{com}$ and $z = \left(\mathsf{ck}, \{d_j\}_{j \in [k]}, \mathsf{op}'\right)$
  - For each $j \in [k]$:
    - For $i \in [\ell]$, set $z_{j,i} \leftarrow \mathsf{TExt}_i(c, d_j)$
    - For $i \in [\ell]$, set $a_{j,i} \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_{j,i})$
  - For each $j \in [k]$, set $\mathbf{v}'_j = (a_{j,1}, \ldots, a_{j,\ell})$
  - Compute and return b as

$$b = \left(\mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}'_j\}_{j \in [k]}, \mathsf{op}')\right) \wedge \left(\bigwedge_{j \in [k], i \in [\ell]} \phi_i(x_i, a_{j,i}, c, z_{j,i}) \right)$$

Figure 15: A compiler for k-out-of-ℓ proofs of partial knowledge.

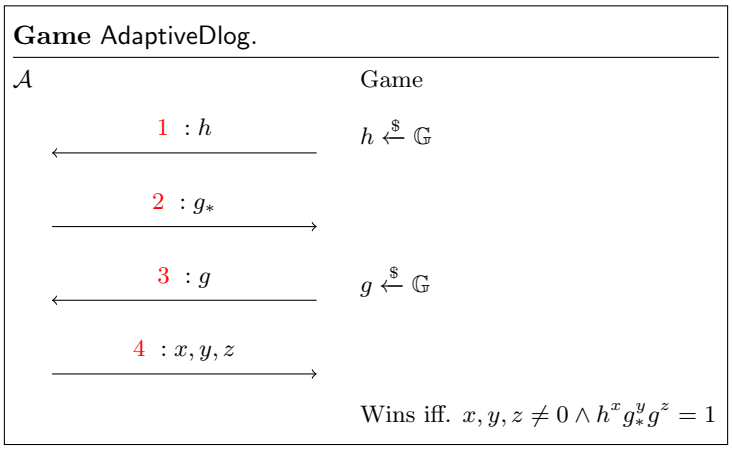

Figure 16: AdaptiveDlog Game. Messages are label (in red) for easy referencing.

Complexity Analysis. Note that we cannot recursively apply this compiler, and so the resulting complexity will be worse than the self-stacking and cross-stacking compilers presented above. In this case, let $|x_{\mathcal{D}}|$ be the size of elements of $\mathcal{D}$. The communication complexity of the protocol produced by applying the compiler in Figure 15 will be $(k|x_{\mathcal{D}}| + |\mathsf{ck}| + |\mathsf{com}| + |\mathsf{op}|)$. Note that the computational complexity of this protocol is $O(k\ell)$, as the prover must simulate or honestly execute all ℓ clauses k times each.

G Optimized Partially Binding Vector Commitments from RO

In this section, we present our optimized construction of partially binding vector commitments. We show that this construction is secure if the discrete log assumption holds. However, showing a direct reduction is cumbersome. Instead, we first formalize a variant of the discrete log assumption, called AdaptiveDlog that is more convenient for our purposes. We will use this variant, presented in Definition 5, as a stepping-stone in our analysis. Intuitively, there are 3 elements in play when an adversary wants to break the construction: an element in the CRS h, the element it gets to choose g_*, and the element corresponding to the index at which it would like to cheat, say g. In order to break the construction, the adversary would somehow need to uncover a relationship in the discreet logs between these values. Note the order in which these are chosen: first the CRS value is sampled, then the adversary selects the value g_* which is not binding. Finally, the random oracle "samples" the remaining group element. AdaptiveDlog captures this game directly; we begin by showing that it is equivalent to the discreet log assumption.

Lemma 6 (Discrete Log reduces to AdaptiveDlog.)**.** *Let $\mathcal{A}$ be an adversary winning the* AdaptiveDlog *game (Figure 16) with probability ϵ, then there exists an expected polynomial-time adversary $\mathcal{A}'$ computing discrete logs (Definition 5) in $\mathbb{G}$ with probability $\geq \epsilon - \mathsf{negl}(\lambda)$.*

Proof. Consider the following PPT algorithm $\mathcal{A}'$:

$y \leftarrow \mathcal{A}'^{\mathcal{A}}(1^\lambda, \mathbb{G}, g, h)$: computes the discrete log y st. $g = h^y$.

Send h to $\mathcal{A}$; $\mathcal{A}$ returns $g_* \in \mathbb{G}$

// Initial query in the h row

$r_1 \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}; g'_1 \leftarrow g^{r_1}$

Send g'_1 to $\mathcal{A}$ (as 'g'); $\mathcal{A}$ returns $(x_1, y_1, z_1) \in \mathbb{Z}^3_{|\mathbb{G}|}$

if $h^{x_1} g_*^{y_1} g^{r_1 z_1} \neq 1 \vee x_1 = 0 \vee y_1 = 0 \vee z_1 = 0$, **return** $\perp$

// Probe the h row without replacement.

$R \leftarrow \{r_1\}; c \leftarrow \top$

while $c = \top \wedge R \neq \mathbb{Z}_{|\mathbb{G}|}$

  Rewind $\mathcal{A}$ to before message 3 (just before sending g)

  $r_2 \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|} \setminus R; g'_2 \leftarrow g^{r_2}$

  Send g'_2 to $\mathcal{A}$ (as 'g'); $\mathcal{A}$ returns $(x_2, y_2, z_2) \in \mathbb{Z}^3_{|\mathbb{G}|}$

  if $h^{x_2} g_*^{y_2} g^{r_2 z_2} = 1 \wedge x_2, y_2, z_2 \neq 0 : c \leftarrow \perp$

  $R \leftarrow R \cup \{r_2\}$

// Extract the discrete log.

Solve the affine system (for free variables α, β):

  $x_1 + y_1\alpha + z_1 r_1 \beta = 0$

  $x_2 + y_2\alpha + z_2 r_2 \beta = 0$

return β

Note that the algorithm recovers $(x_1, y_1, r_1 z_1) \neq (x_2, y_2, r_2 z_2)$ st. $h^{x_1} g_*^{y_1} g^{r_1 z_1} = h^{x_2} g_*^{y_2} g^{r_2 z_2} = 1$ with probability $\epsilon - 1/|\mathbb{G}|$ and with 2 queries to $\mathcal{A}$ in expectation; See Section 3.1 (analysis of the Collision Game) in Attema, et. al [ACK21] for more details. Furthermore since r_1, r_2 are sampled randomly and $\forall i \in [2] : x_i, y_i, z_i \neq 0$, the linear system has full rank except with probability at most $1/|\mathbb{Z}_{|\mathbb{G}|}|$ – which is negligible. Hence $\mathcal{A}'$ recovers the discrete log of g (and g_*) with probability $\epsilon - \mathsf{negl}(\lambda)$. □

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$

$\mathbb{G} \leftarrow \mathsf{GenGroup}(1^\lambda); h \xleftarrow{\$} \mathbb{G}$

return $(\mathbb{G}, h)$

$(\mathsf{com}, \mathsf{aux}) \leftarrow \mathsf{EquivCom}(\mathsf{pp}, \mathsf{ek}, \mathbf{v})$:

$\mathsf{aux} \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$

$\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}, \mathsf{aux})$

return $(\mathsf{com}, \mathsf{aux})$

$\mathsf{com} \leftarrow \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}, r)$:

$g_1 = \mathsf{ck}$

for $i \in [2, \ell] : g_i \leftarrow \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$

return $h^r g_1^{\mathbf{v}_1} g_2^{\mathbf{v}_2} \cdots g_\ell^{\mathbf{v}_\ell}$

$r \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', \mathsf{aux})$:

$g_1 = \mathsf{ck}$

for $i \in [2, \ell] : g_i \leftarrow \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$

$r \leftarrow \mathsf{aux} - \sum_{i \in [\ell]} \mathsf{ek} \cdot (\mathbf{v}'_i - \mathbf{v}_i) \in \mathbb{Z}_{|\mathbb{G}|}$

return r

$(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$

$E = [\ell] \setminus B = \{i_*\}$

$\mathsf{ek} \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}; g_{i_*} \leftarrow h^{\mathsf{ek}}$

// Apply the inverse permutation $i_* - 1$ times to g_{i_*}.

for $i \in [i_* - 1, 1] : g_i \leftarrow \mathsf{P}^{-1}_{|\mathbb{G}|}(g_{i+1})$

$\mathsf{ck} = g_1$

return $(\mathsf{ck}, \mathsf{ek})$

Figure 17: Optimized partially binding vector commitments from discrete log and RO.

Lemma 7. *For a cyclic group $\mathbb{G}$ wherein discrete log is intractable (Definition 5), let $\mathsf{P}_{|\mathbb{G}|} : \mathbb{G} \to \mathbb{G}$ be a cryptographic permutation (modelled as a invertible random oracle) with inverse $\mathsf{P}^{-1}_{|\mathbb{G}|} : \mathbb{G} \to \mathbb{G}$. The construction shown in Figure 17 is a (computationally binding and perfectly hiding) $(\ell-1)$-of-ℓ partially binding vector commitment scheme.*

Proof. The completeness of partial equivocation is easily seen (follows from equivocation of vector Pedersen commitments), so we focus on computational binding and perfect hiding.

Computational Binding Let $\mathcal{A}_k^{\mathsf{P}_{|\mathbb{G}|}}$ be a PPT algorithm winning the binding game with probability ϵ i.e.

$$\epsilon = \Pr\left[\begin{matrix}\nexists S \subset [\ell], |S| \geq t, \text{ s.t. } i \in S, v_{1,i} = \ldots = v_{k,i} \ \wedge \\ \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}_1; r_1) = \ldots = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}_k; r_k)\end{matrix} \middle| \begin{matrix}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); \\ (\mathsf{ck}, \mathbf{v}_1, \ldots, \mathbf{v}_k, r_1, \ldots, r_k) \leftarrow \mathcal{A}_k^{\mathsf{P}_{|\mathbb{G}|}}(1^\lambda, \mathsf{pp})\end{matrix}\right]$$

Then $\mathcal{A}'$ (Figure 18) wins the $\mathsf{AdaptiveDlog}$ game with probability $\epsilon' = \epsilon/\mathsf{poly}(\lambda) - \mathsf{negl}(\lambda)$. We lower bound the probability that $h^{\hat{x}} g_*^{\hat{y}} g^{\hat{z}} = 1$ and $\hat{x}, \hat{y}, \hat{z} \neq 0$. Start by observing that in the chain: $\forall i \in [2, \ell]$: $g_{i+1} = \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$, there can be at most one group element g'_* on which the oracle is queried, but which has not been output by $\mathsf{P}_{|\mathbb{G}|}$: there are $\ell - 1$ outputs and ℓ group elements. If all elements in the chain has been output by $\mathsf{P}_{|\mathbb{G}|}$, then define $g'_* = g_1$. Suppose the reduction guesses correctly and $g_* = g'_*$, this occurs with noticeable probability $1/\mathsf{poly}(\lambda)$. Suppose furthermore that $\mathcal{A}$ wins the binding game, in this case we know: $\Pr_\delta[\mathsf{HW}(\mathbf{w}) \geq 3] = 1 - 1/|\mathbb{G}|$, because $\mathsf{HW}(\mathbf{w}^{(1)}) \geq 2, \mathsf{HW}(\mathbf{w}^{(2)}) \geq 2$ and $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}$ have at least one distinct non-zero position each. Note additionally that $\mathbf{g}^{\mathbf{w}} = \mathbf{g}^{\mathbf{w}^{(1)}} \cdot (\mathbf{g}^{\mathbf{w}^{(2)}})^\delta = 1 \cdot 1^\delta = 1$, furthermore:

$$\mathbf{g}^{\mathbf{w}} = \prod_{i \in [\ell] \cup \{0\}} \mathbf{g}_i^{\mathbf{w}_i} = \prod_{i \in [\ell] \cup \{0\}, (x,y,z) = W_i} (h^x g_*^y, g^z)^{\mathbf{w}_i} = g^{\hat{x}} g_*^{\hat{y}} g^{\hat{z}} = 1$$

To bound the probability that $\hat{x}, \hat{y}, \hat{z} \neq 0$, observe that $\exists j \in [\ell] \setminus \{i_*\}$ where $\mathbf{w}_j \neq 0$ (since $\mathsf{HW}(\mathbf{w}) \geq 3$). Hence $\hat{x}, \hat{y}, \hat{z}$ can be expressed as: $\hat{x} = \hat{x}' + \mathbf{w}_j x$, $\hat{y} = \hat{y}' + \mathbf{w}_j y$, $\hat{z} = \hat{z}' + \mathbf{w}_j z$, where x, y, z are sampled i.i.d. uniform (since $j \notin \{0, i_*\}$). Hence the probability that either of $\hat{x}, \hat{y}, \hat{z}$ are zero, is at most $3/|\mathbb{G}|$ by a union bound.

Perfect Hiding Simply observe that for any permutation $P : \mathbb{G} \to \mathbb{G}$, the distribution $\{P(g) \mid g \xleftarrow{\$} \mathbb{G}\}$ is uniform. Therefore the distribution of ck is the same for any $B = \{i_*\}$ (by letting $P = \mathsf{P}^{-(i_*-1)}_{|\mathbb{G}|}$; repeated applications of $\mathsf{P}^{-1}_{|\mathbb{G}|}$ $(i_* - 1)$ times).

□

$\mathcal{A}'^{\mathcal{A}_k^{\mathsf{P}_{|\mathbb{G}|}}}(1^\lambda, \mathbb{G})$ plays the $\mathsf{AdaptiveDlog}$ game.

1 : Receive h from the $\mathsf{AdaptiveDlog}$ game.

2 : Sample $\hat{q} \xleftarrow{\$} [\mathsf{poly}(\lambda)]$ where $\mathsf{poly}(\lambda)$ is a bound on the number of RO queries made by $\mathcal{A}$

3 : Run $(\mathsf{ck}, \mathbf{v}^{(1)}, \ldots \mathbf{v}^{(k)}, r_1, \ldots, r_k) \leftarrow \mathcal{A}_k^{\mathsf{P}_{|\mathbb{G}|}}(1^\lambda, \mathsf{pp} = (\mathbb{G}, h))$

Whenever $\mathcal{A}_k^{\mathsf{P}_{|\mathbb{G}|}}$ makes the q'th query to $\mathsf{P}_{|\mathbb{G}|}$ (or $\mathsf{P}_{|\mathbb{G}|}^{-1}$) on (previously unprogrammed) $Q_q \in \mathbb{G}$:

a : **if** $q < \hat{q} : R_q \xleftarrow{\$} \mathbb{G}$; **return** R_q

b : **if** $q = \hat{q}$:

A : Let $g_* = Q_q$

B : Send g_* to $\mathsf{AdaptiveDlog}$; Receive g from $\mathsf{AdaptiveDlog}$.

c : **if** $q \geq \hat{q}$:

A : $x_q, y_q, z_q \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$

B : $R_q \leftarrow h^{x_q} g_*^{y_q} g^{z_q} \in \mathbb{G}$

C : **return** R_q

4 : Compute $g_1 = \mathsf{ck}$; **for** $i \in [2, \ell] : g_i \leftarrow \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$

5 : **if** $\nexists i_* \in [\ell] : g_i = g_* :$ **return** $\perp$ (reduction "guessed $\hat{q}$ wrong")

6 : **for** $i \in [\ell] \setminus \{i_*\} : W_i = (x_q, y_q, z_q)$ st. $\exists q : R_q = g_i$

7 : Let $W_0 = (1, 0, 0), W_{i_*} = (0, 1, 0)$

8 : Pick $p_1, p_2 \in [\ell]$, with $p_1 \neq p_2$ st.

$$\exists i_1, i_1' : \mathsf{com}_1 = \mathsf{BindCom}(\mathsf{p}, \mathsf{ck}, \mathbf{v}^{(i_1)}, r_{i_1}) = \mathsf{BindCom}(\mathsf{p}, \mathsf{ck}, \mathbf{v}^{(i_1')}, r_{i_1'}) \wedge \mathbf{v}_{p_1}^{(i_1)} \neq \mathbf{v}_{p_1}^{(i_1')}$$

$$\exists i_2, i_2' : \mathsf{com}_2 = \mathsf{BindCom}(\mathsf{p}, \mathsf{ck}, \mathbf{v}^{(i_2)}, r_{i_2)}) = \mathsf{BindCom}(\mathsf{p}, \mathsf{ck}, \mathbf{v}^{(i_2')}, r_{i_2'}) \wedge \mathbf{v}_{p_2}^{(i_2)} \neq \mathbf{v}_{p_2}^{(i_2')}$$

If no such p_1, p_2 exists: **return** $\perp$ (note $\mathcal{A}_k^{\mathsf{P}_{|\mathbb{G}|}}$ loses the game)

9 : Define:

$$\mathbf{w}^{(1)} \coloneqq (r_{i_{p_1}} \| \mathbf{v}^{(i_{p_1})}) - (r_{i'_{p_1}} \| \mathbf{v}^{(i'_{p_1})}) \in \mathbb{Z}_{|\mathbb{G}|}^{\ell+1}$$

$$\mathbf{w}^{(2)} \coloneqq (r_{i_{p_2}} \| \mathbf{v}^{(i_{p_2})}) - (r_{i'_{p_2}} \| \mathbf{v}^{(i_{p_2})'}) \in \mathbb{Z}_{|\mathbb{G}|}^{\ell+1}$$

$\mathbf{g} \coloneqq (h, g_1, \ldots, g_\ell) \in \mathbb{G}^{\ell+1}$. Note: $\mathbf{g}^{\mathbf{w}^{(1)}} = 1 \in \mathbb{G}, \mathbf{g}^{\mathbf{w}^{(2)}} = 1 \in \mathbb{G}$

10 : Pick $\delta \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$, define: $\mathbf{w} = \mathbf{w}^{(1)} + \delta \cdot \mathbf{w}^{(2)}$

11 : Let:

$$\hat{x} = \textstyle\sum_{i \in [\ell] \cup \{0\}, (x,y,z) = W_i} \mathbf{w}_i \cdot x$$

$$\hat{y} = \textstyle\sum_{i \in [\ell] \cup \{0\}, (x,y,z) = W_i} \mathbf{w}_i \cdot y$$

$$\hat{z} = \textstyle\sum_{i \in [\ell] \cup \{0\}, (x,y,z) = W_i} \mathbf{w}_i \cdot z$$

12 : Send $(\hat{x}, \hat{y}, \hat{z})$ to the $\mathsf{AdaptiveDlog}$ game.

Figure 18: Reduction for partially binding commitment scheme to discrete log in the programmable, invertible random oracle model.