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

A sequence of recent works by Heath and Kolesnikov have explored modifying existing interactive protocols for privacy-preserving computation (secure multiparty computation, private function evaluation and zero-knowledge proofs) to be more communication efficient when applied to disjunctive statements, such that the cost only depends on the size of the largest clause in the disjunction.

In this work, we focus on the specific case of zero-knowledge proofs for disjunctive statements. 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 and Guillou-Quisquater) and modern MPC-in-the-head protocols such as the recent work of Katz, Kolesnikov and Wang and the Ligero protocol of Ames *et al.* 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 non-interactive zero-knowledge protocol for disjunctions where the communication only depends on the size of the largest clause.

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, 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 in 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]), 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 parallel 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, HK20a]. These techniques have also been extended to reducing the communication complexity of secure computation protocols containing conditional clauses [HK20a, HKP20a].

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 privacy-free nature of the 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. Indeed, completely new garbled circuit stacking techniques are required to reduce the communication complexity of interactive garbling [HK20a].

In this work, we take a more general approach to 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 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].

Benefits of a Generic Stacking Compiler. There are several significant benefits to developing generic stacking compilers, rather than developing bespoke protocols that support stacking. First, automatically compiling multiple Σ-protocols into one that supports 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 stacking. For instance, a protocol designer can select a proof technique that naturally operates over the most efficient representation of their statement (*e.g.* boolean circuit, arithmetic circuit, elliptic curve, algebraic structure), even if there is no known modification of the proof technique to support stacking. This removes the significant overheads incurred when rephrasing statements or reducing statements to some other NP complete language. This is particularly relevant, as 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, HK20a] 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 [1] and flattening the resulting protocol using the Fiat-Shamir [FS87] transform yields in a ring signature construction with signatures of length $\lambda \cdot (2+2\log(\ell))$ 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 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 and preserves the post-quantum security 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 Σ be a stackable Σ-protocol for an NP language $\mathcal{L}$ that has communication complexity $\mathtt{CC}(\Sigma)$. 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 $O(\mathtt{CC}(\Sigma) + \lambda \log(\ell))$, where λ is the 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 Σ_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 $O(f_{\mathtt{CC}}(\{\Sigma_i\}_{i\in[\ell]}) + \lambda \log(\ell))$.*

Examples of Stackable Σ-protocols. We show concrete examples of Σ-protocols that are stackable. Specifically, we look at classical protocols like Schnorr [Sch91], 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 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 $O(\sqrt{|C|} + \log \ell)$.

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

[1] using the DDH-based partially binding commitment scheme in the random oracle model we describe later in this work

can be constructed in the random oracle model from any non-interactive equivocal commitments with commitments keys that are uniformly random bitstrings. An interactive variant of a similar notion was recently used in [BMRS20]. Our compiler can also work with their notion to yield protocols in the plain model.

Concretely Efficient Σ-Protocols for Disjunctions. Our compiler can be concretely instantiated with very small additive overhead that is logarithmic in the number of clauses. For instance, if we instantiate our partially binding vector commitments based on Pederson commitments in the random oracle model, the compiler adds only $(2\lambda \log(\ell))$ bits of overhead! We also show how to instantiate our partially binding vector commitments from any equivocal commitment scheme, which yields slightly higher overhead but can also be done from plausibly post-quantum secure assumptions. Random oracles can also be avoided at the cost of necessitating interactivity using the interactive commitment scheme of [CPS$^+$16]

Future Work. In this work we focus on Σ-protocols for ease of explication and to capture a wide class of intresting 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 $|\pi| = O(\ell)$. Groth and Kohlweiss [GK15] constructed a zero-knowledge proof of partial knowledge for the "discrete log" relation i.e. $\mathcal{R}_1 = \ldots = \mathcal{R}_\ell = \mathcal{R}_{\text{dlog}} \coloneqq x \overset{?}{=} g^w$ with threshold $t = 1$ and communication $|\pi| = O(\log \ell)$. Later work by Attema and Cramer [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. Their instantiation of '(k, n)-equivocal commitments' allows the use of our compiler without random oracles.

Free IF. Work by Kolesnikov [Kol18] showed how to construct S-universal two party SFE (Secret Function Evaluation), from Yao's garbled circuits [Yao86] and Oblivious Transfer (OT): this is achieved by introducing *topology-decoupling circuit garbling*, which is the observation that garbled circuits for different circuits with the same number of AND gates look indistinguishable. This is exploited by having the generator garbling the active clause f_a, then for every function $f \in S$ the evaluator evaluates the garbled circuit as if it was a garbling of f, for the active branch f_a he obtains valid labels, for all others he obtains 'junk', finally an output selection protocol is run to only obtain the output of f_a. The idea of viewing garbled circuits as 'just a bunch of bits' lead to a line of work:

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 branch seperately 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].

MOTIF. Heath et al. [HKP20b] shows how to execute multiple branches in parallel with GMW without increasing the number of OTs (Oblivious Transfers). [HKP20b] observes that (Boolean) scalar/vector (of dimension k) multiplication can be implemented using a 1-of-2 OT over k bit strings, which enables evaluating the circuits for each of the k branches in parallel. Asymptotically the communication complexity of evaluating k branches of size $|C|$ is $O(kn^2|C|)$ and relies on branches having similar topology for AND gates, however the performance increase is very substantial[2] in practice. Since oblivious transfers 'come for free'[3] when doing MPC-in-the-head, it is not immediate how this technique can be used to improve communication of GMW-style IKOS derived proofs e.g. ZKB++ [CDG+17].

Mac'n'Cheese. Concurrent work by Baum et. al [BMRS20] introduces an abstraction dupped 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 achieve a generic framework to achieve communication-efficient disjunctions of Σ-protocols without requiring non-trivial[4] 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 each clause uses the same Σ-protocol to prove a different statement. We will then extend our ideas to cover heterogeneous Σ-protocols.

We start by considering the approaches taken by recent work focusing on privacy-preserving protocols for disjunctive statements, *e.g.* [HK20b,HK20a]. 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 adversary 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. The intuition extracted from prior work leads us to a natural high-level template to achieve 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 transcript of a Σ-protocol can be safely re-used? We now discuss these challenges, and the techniques we use to overcome them, in more detail.

[2]Going from roughly λk to $k + \lambda$ communication per AND gate.

[3]The communication cost is that of adding the output to the receivers view.

[4]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." We discuss this in the context of MPC-in-the-head protocols in Section 6.

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 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 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). As such, we require a new approach for cheating on inactive clauses.

Towards this, 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 *non-interactive, partially-binding, vector commitments*.[5] These commitments allow the committer to commit to a vector of values and equivocate on some of the vector entries later. For instance, a 1-out-of-ℓ binding commitment allows the committer to commit a vector of ℓ values in a way 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 along with $\ell - 1$ garbage values using the 1-out-of-ℓ binding commitment and sends this commitment to the verifier.
- **Round 2:** The verifier sends challenge messages for each of the ℓ instances.
- **Round 3:** The prover honestly computes a third round message for the active clause and simulates the 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 show several practical constructions. First, we show how to build this primitive from any non-interactive equivocal commitment scheme in the random oracle model. This results in concretely efficient instantiations and can also be made plausibly post-quantum secure. Additionally, we note that the DDH based commitment scheme from [CPS+16] can also be used, facilitating constructions in the plain model.

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 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 round message.* That is, it is often easy to generate an accepting transcript for a chosen challenge and third round message. This allows the prover to create a transcript for the inactive clauses that shares a 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 does its simulation *with respect to the third round message of the active transcript.* This allows

[5]A similar notion for interactive commitments was introduced in [CPS+16]. Note that this notion of commitments is very different from the notion of somewhere statistically binding commitments [FLPS20].

the prover to send a single third round message that can be re-used for 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 — 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-protocol. 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.[6] 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.

However, many natural Σ-protocols satisfy both properties. 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. If the simulator computes the first round messages deterministicallly, 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. 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 w.l.o.g. using a hash function.

Summary of our Stacking Compiler. Having now outlined our techniques, we present a detailed example of our compiler for 2 clauses, as depicted in Figure 1. 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' to open the commitment a' to the vector $\mathbf{v}' = (a_1, a_2)$. (8) The compiled third round message consists of the honestly generated third round message z, and the randomness r' of the equivocated commitment. (9) Given the third round message z and the challenge messages, the verifier first recomputes the two first round messages a_1, a_2.[7]. The verifier then verifies the proof by ensuring that each transcript is accepting and that the first round messages constitute a valid opening of the commitment a'.

Complexity Analysis: As noted above, the first round message in the underlying Σ-protocol does not depend on the size of the relation circuit. Therefore, the communication in the first round is $O(\ell\lambda)$, where λ is the security parameter. In the last round, the prover sends one third round message of the underlying

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

[7]Note that this can be done, because we assume that given a third round message and a challenge, our simulator deterministically computes a unique first round message.

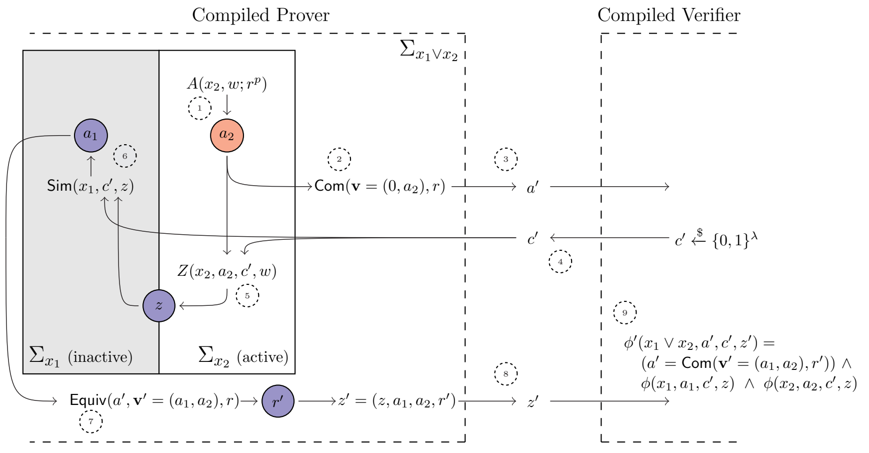
Compiled Prover
Compiled Verifier
$\Sigma_{x_1 \vee x_2}$
$A(x_2, w; r^p)$
a_1
a_2
$\mathsf{Sim}(x_1, c', z)$
$\mathsf{Com}(\mathbf{v} = (0, a_2), r)$
a'
c'
$c' \xleftarrow{\$} \{0,1\}^\lambda$
$Z(x_2, a_2, c', w)$
z
Σ_{x_1} (inactive)
Σ_{x_2} (active)
$\phi'(x_1 \vee x_2, a', c', z') =$
$(a' = \mathsf{Com}(\mathbf{v}' = (a_1, a_2), r')) \wedge$
$\phi(x_1, a_1, c', z) \wedge \phi(x_2, a_2, c', z)$
$\mathsf{Equiv}(a', \mathbf{v}' = (a_1, a_2), r)$
r'
$z' = (z, a_1, a_2, r')$
z'
1
2
3
4
5
6
7
8
9

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 for readability. 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.

Σ-protocol that depends on the size of one of the clauses[8] and ℓ first round messages of the underlying Σ-protocol. Thus, communication in the third round is $O(\mathsf{CC}(\Sigma) + \ell \cdot \lambda)$, where $\mathsf{CC}(\Sigma)$ is the communication complexity of the underlying stackable Σ-protocol, when executed for the largest clause. We further show that our resulting protocol is also "stackable" and can be recursively used to reduce the communication complexity to $O(\mathsf{CC}(\Sigma) + \log(\ell) \cdot \lambda)$, where $\mathsf{CC}(\Sigma)$

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 like KKW [KKW18] and Ligero [AHIV17] have these properties. In fact, we show that any Σ-protocol in MPC-in-the-head paradigm [IKOS07] that is based on an MPC protocol with a *special property*, will satisfy the properties. 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 *i.e.* $(x_1 \in \mathcal{L}) \vee \ldots \vee (x_\ell \in \mathcal{L})$. 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 compiler in a straightforward way. For instance, one could create a 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 approach; 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 may 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

[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.
[9]While it might be possible to define a Σ-protocol that uses different techniques for different parts of the relation, this would

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, 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].

The above ideas can also be extended to obtain efficient zero-knowledge proofs for statements where the prover wishes to prove that it has witnesses to at least k out of the ℓ clauses. Due to space constraints, we are unable to discuss this in the main body of the work, but include it in the supplementary material (Appendix G). The communication complexity of the resulting protocol depends on k times the cost of running a single instance of the most communication-heavy Σ-protocol.

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 and cross-stacking compilers in Section 7 and Section 8 respectively.

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.

4 Preliminaries

4.1 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 for $\mathcal{R}$ is a 3 move protocol between a prover* P *and a verifier* V *consisting of a tuple of PPT algorithms (A, Z, ϕ) 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.*
- $c \overset{\$}{\leftarrow} \{0,1\}^\lambda$*:* V *samples a random challenge.*
- $z \leftarrow Z(x, w, c; r^p)$*: 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 (A, Z, ϕ) is said to be complete if for any x, w such that $\mathcal{R}(x, w) = 1$, and any $r^p \overset{\$}{\leftarrow} \{0,1\}^\lambda, r^v \overset{\$}{\leftarrow} \{0,1\}^\lambda$, it holds that,*

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

- ***Special Soundness.*** *A Σ-Protocol (A, Z, ϕ) 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 (A, Z, ϕ) 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 $\mathcal{R}(x, w) = 1$, and any $r^v \overset{\$}{\leftarrow} \{0,1\}^\lambda$, it holds that*

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

4.2 Secure Multiparty Computation

For completeness, we recall the definitions of t-privacy and t-robustness from [IKOS07].

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.* These commitments allow a commiter to commit to a vector of ℓ elements such that exactly t positions are binding (*i.e.* cannot be open to another value) and the remaining $\ell - t$ positions can be equivocated. The commiter 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* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Com}, \mathsf{Equiv})$ *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.
- $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r)$*: Takes public parameter* pp*, commitment key* ck*, ℓ-tuple $\mathbf{v}$ and randomness r. Returns a partially binding commitment.*
- $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', r)$*: Takes public parameters* pp*, equivocation key* ek*, original commitment value $\mathbf{v}$ and updated commitment values $\mathbf{v}'$ with $\forall i \in B : \mathbf{v}_i = \mathbf{v}'_i$. Returns an opening r' of the commitment $c = \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r)$ to the value $\mathbf{v}'$*

For simplicity of presentation, we assume that the Open *function recomputes the commitment, given the vector and randomness to check for correctness and hence omit it from our interface. The properties satisfied by the above algorithms are as follows:*

(Perfect) Hiding: *The commitment key* ck *(perfectly) hides the binding positions B and commitments c (perfectly) hides the committed ℓ values in $\mathbf{v}$. More formally:*

$$\forall \mathbf{v}, \mathbf{v}' \in \mathcal{M}^\ell,\ \forall B, B' \in \binom{[\ell]}{t} :$$

$$\begin{aligned} &\left[(\mathsf{ck}, \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r)) \mid (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B); r \leftarrow \{0,1\}^n; \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)\right] \\ \overset{p}{\equiv}\ &\left[(\mathsf{ck}', \mathsf{Com}(\mathsf{pp}, \mathsf{ck}', \mathbf{v}', r)) \mid (\mathsf{ck}', \mathsf{ek}') \leftarrow \mathsf{Gen}(\mathsf{pp}, B'); r \leftarrow \{0,1\}^n; \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)\right] \end{aligned}$$

(Computational) Partial Binding: *It is intractable to find a commitment key* ck*, ℓ-tuples $\mathbf{v}, \mathbf{v}'$ and randomness r, r' such that $\mathbf{v}$ and $\mathbf{v}'$ differ in more that $\ell - t$ positions and have the same commitment. More formally, for all PPT algorithms $\mathcal{A}$:*

$$\Pr\left[D_{Ham}(\mathbf{v}, \mathbf{v}') > \ell - t \ \wedge\ \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r) = \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}'; r') \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); \\ (\mathsf{ck}, \mathbf{v}, \mathbf{v}', r, r') \leftarrow \mathcal{A}(1^n, \mathsf{pp}) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

Where $D_{Ham}(\mathbf{v}, \mathbf{v}')$ is the Hamming distance.

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 $\forall i \in B : \mathbf{v}_i = \mathbf{v}'_i$. More formally:*

$$\forall\ B \in \binom{[\ell]}{t},\ \forall\ \mathbf{v}, \mathbf{v}' \in \mathcal{M}^\ell \text{ st. } \forall i \in B : \mathbf{v}_i = \mathbf{v}'_i \text{ then:}$$

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

We additionally require the size of the commitment to only be linear in the security parameter. We note that this is easy to achieve w.l.o.g. using a hash function.

We explore constructions of partially binding commitments from different assumptions in Appendix A.

6 Stackable Σ-Protocols

In this section, we present the properties of Σ-protocols that our stacking framework requires, showing 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

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 possible third round messages, and then constructing the unique appropriate first round message. We refer to such a simulator as an *extended simulator.*

Definition 5 (EHVZK Σ-Protocol) *Let $\Sigma = (A, Z, \phi)$ be a Σ-protocol for the NP relation $\mathcal{R}$. We say that Σ 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 $c \in \{0,1\}^\lambda$, there exists an efficiently samplable distribution $\mathcal{D}^{(z)}_{x,c}$ such that:

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

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, then finding the first round message a which 'matches' without using 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 that are not EHVZK: consider a 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. We also note in Observation 1 that any Σ-protocol into one which is EHVZK.

Observation 1 *Any Σ-protocol $\Sigma = (A, Z, \phi)$ can be transformed into an EHVZK Σ-protocol $\Sigma' = (A', Z', \phi')$ for the same relation as follows:*

1. *$a' \leftarrow A'(x, w; r^p)$: Compute $a' \leftarrow A(x, w; r^p)$.*
2. *$z' \leftarrow Z(x, w, c'; r^p)$: Compute $z \leftarrow Z(x, w, c'; r^p)$ and output $z' = (a, z)$.*
3. *$b' \leftarrow \phi'(x, a', c', z')$: Parse $z' = (a, z)$. Output $a \stackrel{?}{=} a' \wedge \phi(x, a, c', z)$.*

It is obvious that the protocol above is EHVZK by letting $\mathcal{D}^{(z)}_{x,c}$ be the output distibution of $\mathcal{S}_\Sigma(x, c)$ (the Special Honest-Verifier Zero-Knowledge simulator of Σ) and letting $\mathcal{S}^{\text{EHVZK}}(z' = (a, z)) := a$.

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 all 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 6 (Σ-Protocol with Recyclable Third Messages) *Let $\mathcal{R}$ be an NP relation and $\Sigma = (A, Z, \phi)$ be a Σ-protocol for $\mathcal{R}$. We say that Σ has* recyclable third messages *if for each $c \in \{0,1\}^\lambda$, there exists an efficiently sampleable distribution $\mathcal{D}_c^{(z)}$, such that for all instance-witness pairs (x, w) st. $\mathcal{R}(x, w) = 1$, it holds that*

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

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.[10] Although this property might seem strange, we will later show that many natural Σ-protocols have this property.

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

Definition 7 (Stackable Σ-Protocol) *We say that a Σ-protocol $\Sigma = (A, Z, \phi)$ is* stackable*, if it is EHVZK (see Definition 5) and has recyclable third messages (see Definition 6).*

We now note a useful property of stackable Σ-protocols that follow directly from Definition 7:

Remark 1 *Let $\Sigma = (A, Z, \phi)$ be a* stackable *Σ-protocol for the NP relation $\mathcal{R}$. 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,*

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

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.

Lemma 1 *Schnorr's Σ-Protocol [Sch91] is stackable.*

Proof 1 *Recall Schnorr's Σ-protocol for the relation $\mathcal{R}_{\mathbb{G},g}(x, w) := x \overset{?}{=} g^w$, where $\langle g \rangle = \mathbb{G}$ is a cyclic group and $w \in \mathbb{Z}_{|\mathbb{G}|}$. For $r^p, r^v \overset{\$}{\leftarrow} \mathbb{Z}_{|\mathbb{G}|}$, the protocol proceeds as follows: (1) $A(x, w; r^p) : g^{r^p} = a$ (2) $Z(x, w, c; r^p) : cw + r^p = z$ (3) $\phi(x, a, c, z) : g^z \overset{?}{=} x^c a$. We now show that this protocol is stackable.*

Let the extended simulator be $\mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z) := g^z x^{-c}$. Given a statement $x \in \mathcal{L}_g$ and $c \in \mathbb{Z}_{|\mathbb{G}|}$, let $\mathcal{D}_{x,c}^{(z)}$ be the uniform distribution on $\mathbb{Z}_{|\mathbb{G}|}$. Observe that the prover messages (a, z) in both real and simulated transcripts are uniformly random subject to the constraint $g^z = x^c a$. Hence this protocol is EHVZK. Moroever, since the above distribution $\mathcal{D}_{x,c}^{(z)}$ does not depend on x (or even c), this protocol also has recyclable third messages.

We also show that Guillou-Quisquater's Σ-protocol [GQ90] and Blum's 3 move protocol [Blu87] for graph hamiltonicity are stackable. The protocols and proofs for those can be found in the Appendix in Section C.

Lemma 2 *Guillou-Quisquater's Σ-protocol [GQ90] is stackable.*

Lemma 3 *Blum's Σ-protocol [Blu87] is stackable.*

[10]We further elaborate on this in Remark 1.

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 naturally achieve 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[11] 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.

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

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

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. Although naïvely re-interpreting a transcript in this way may lead to inconsistencies when verifying the transcript; this can be handled by considering a slightly modified version of the 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: $\mathsf{ExecuteMPC}$ emulates execution of the protocol on a given function with virtual parties and outputs the actual views of the parties, $\mathsf{CondenseViews}$ takes the views of a subset of the parties as input and outputs their condensed views, and $\mathsf{ExpandViews}$ 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 $\mathsf{ExecuteMPC}$ 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 $\mathsf{CondenseViews}$ 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 $\mathsf{ExpandViews}$ 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 1 (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 8 ($\mathcal{F}$-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 now proceed to show that when instantiated with an $\mathcal{F}$-universally simulatable MPC protocol that has, Theorem 1 yields a stackable Σ-protocol.

Theorem 2 ($\mathcal{F}$-universally simulatable implies $\mathcal{F}$-stackable) *The IKOS compiler (see Theorem 1) yields an stackable* Σ*-protocol when instantiated with an* $\mathcal{F}$*-universally simulatable MPC protocol (see Definition 8) with privacy and robustness (See Definitions 2,3) against a subset of the parties.*

Proof 2 (Theorem 2) *We define the distribution* $\mathcal{D}^{(z)}_{x,c}$, *where* $c \in \{0,1\}^\lambda$ *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* $c \in \{0,1\}^\lambda$ *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 8) 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 8) 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 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,* $\mathcal{D}^{(z)}_{c}$ *is the same as* $\mathcal{D}^{(z)}_{x,c}$ *and this protocol is indeed stackable.*

We now use Theorem 2 to show that two popular IKOS-based Σ-protocols are stackable, namely KKW [KKW18] and Ligero [AHIV17]. The intuition is very similar to that present 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 the Appendix in Sections D.4 and E respectively.

Lemma 4 (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 5 (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 be clauses that the prover did not select. For instance, when it may be possible for an adversarial party to have selected several of the clauses. Thus, for the sake of generality, there can be no guarantee that the all the clauses in the disjunction are actually true instances, and our compilation framework should be applicable to the case where some of the instances are false. At first glance, this might seem like a strange concern: for most interesting NP languages of interest, it should be hard to tell if an instance is in the language, and therefore having false instances in the disjunction should not be a problem.

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 could 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-negligable probability in two cases: (1) the input instance is trivially false (*e.g.*

a connected graph is not 3-colorable), or (2) the simulator has a hardcorded 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 *correctness* 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 to a verifier does not undermine zero-knowledge: the verifier already could check that this clause was a false instance simply by running the simulator algorithm on the clause independently. As such, if the simulator outputs non-accepting transcripts with some non-negligable probability, an adversary could detect this failure case in polynomial time. However, if the verification algorithm allows some transcript to be non-accepting, a malicious prover could trivially exploit this property to violate soundness.

We emphasize that this is a corner case: commonly constructed simulators will in most likelyhood produce accepting transcripts on false instances unless the instance is trivially false or not in the domain of the simulator. 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 9 (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 statements 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 Section B.

Lemma 6 (Simulators are well-behaved without loss of generality) *Every Σ-protocol Σ can be converted to a Σ-protocol Σ' for the same relation with a well-behaved simulator. Furhermore, if Σ' is EHVZK then Σ' is also EHVZK and if Σ has recyclable third messages then Σ' 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. By self-stacking, we mean a compiler that takes a *stackable* sigma protocol Σ 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 sigma 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 3 (Self-Stacking) *Let $\Sigma = (A, Z, \phi)$ be a stackable (See Definition 7) Σ-protocol for the NP relation $\mathcal{R} : \mathcal{X} \times \mathcal{W} \to \{0,1\}$ and let $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Com}, \mathsf{Equiv})$ be a 1-out-of-ℓ binding vector commitment scheme (See Definition 4). For any $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, the compiled protocol $\Sigma' = (A', Z', \phi')$ described in Figure 2 is a* <u>*stackable*</u> *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 3 *We now prove that the protocol $\Sigma' = (A', Z', \phi')$ described in Figure 2 is a stackable Σ-protocol for the relation $\mathcal{R}'((x_1, \ldots, x_\ell), (\alpha, w)) := \mathcal{R}(x_\alpha, w)$.*

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 Σ' using the extractor $\mathcal{E}$ for the underlying sigma protocol Σ. The extractor $\mathcal{E}'$ is given two accepting transcripts for the protocol Σ' 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 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 Σ 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). *We denote distribution of third round message for Σ' 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 third round message z from the distribution of third round messages of the underlying protocol Σ, denoted $\mathcal{D}_c^{(z)}$. Note that by the hiding property of the commitment scheme, the distribution of ck is independent of the binding index B. More formally, for $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$,*

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

Note that this distribution is independent of the statements, as $\mathcal{D}_c^{(z)}$ 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 commitment key ck and randomness r:

$$\begin{aligned} {\mathcal{S}^{\text{EHVZK}}}'((x_1, \ldots, x_\ell), c, (\mathsf{ck}, r, z)) :=& \\ & \forall i \in [\ell] \text{ compute } a_i \leftarrow \mathcal{S}^{\text{EHVZK}}(x_i, c, z) \\ & \textbf{return } a' = \mathsf{Com}(\mathsf{ck}, \mathbf{v} = (a_1, \ldots, a_\ell); r) \end{aligned}$$

Let $\mathcal{D}^{(\alpha,w)}$ denote the distribution of transcripts resulting from an honest prover possessing witness (α, w) running Σ' 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 Σ, $\mathcal{H}^{(\alpha)} \approx \mathcal{D}^{(\alpha,w)}$. 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 $\mathcal{H}^{(\alpha,\mathsf{ck})} \overset{p}{=} \mathcal{H}^{(\alpha)}$ by the (perfect) hiding of the partially binding commitment scheme. Lastly note that $\mathcal{H}^{(\alpha,\mathsf{ck})}$ matches the output distribution of ${\mathcal{S}^{\text{EHVZK}}}'((x_1, \ldots, x_\ell), c, {\mathcal{D}_c^{(z)}}')$.*

Therefore Σ' is a stackable Σ-protocol.

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

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 $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r)$.
  - 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 $Z'(x, w, c; r^p) \to z$ 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 $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', r)$
  - Send $z = (\mathsf{ck}, z^*, r')$ to the verifier.
- **Verification:** Verifier computes $\phi'(x, a, c, z) \to b$ as follows:
  - Parse $a = \mathsf{com}$ 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:

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

Figure 2: A compiler for stacking multiple instances of a Σ-protocol.

Complexity Analysis. Let $\mathtt{CC}(\Sigma)$ be the communication complexity of Σ. Then, the communication complexity of the Σ' obtained from Theorem 3 is $(\mathtt{CC}(\Sigma) + \ell O(\lambda) + |\mathsf{com}| + |r'|)$, where the sizes of com, and r' are all dependent on the choice of partially binding vector commitment scheme and are independent of $\mathtt{CC}(\Sigma)$ and we assume that the size of the commitment key is approximately $\ell O(\lambda)$. If we use Pederson-based, concretely efficient partially binding vector commitment, as discussed in Section 5, then the communication complexity of Σ' is $(\mathtt{CC}(\Sigma) + \ell O(\lambda))$. We explore other choices in the Appendix in A.3. We make use of the commitment scheme in a non-interactive way; however, an interactive commitment scheme can also be used and executed in parallel with Σ'. In general, the computation complexity of this protocol is ℓ times that of Σ. 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.

However, care should be take to ensure that this does not leave the protocol vulnerable to timing attacks.

Optimizations and Extensions. Since our resulting protocol Σ' is also stackable, the communication complexity can be reduced further to $(\mathsf{CC}(\Sigma) + 2\log(\ell)O(\lambda) + |\mathsf{com}| + |r'|)$ by making a recursive use of our compiler as follows: let Σ_2 be the outcome of stacking Σ twice, Σ_4 be the outcome of stacking Σ_2 twice and so on. For ℓ clauses, we only need to stack $\Sigma_{\ell/2}$ twice. It is easy to see that complexity of each Σ_i is $(\mathsf{CC}(\Sigma) + 2\log(i)O(\lambda) + |\mathsf{com}| + |r'|)$.

Moreover, note that in general the stackability property of any Σ-protocol is preserved under parallel execution. Therefore, if $\{a_i\}_{i\in[\ell]}$ are not sent in the third round, our compiler can also be used to get Σ protocols for statements of the form

$$((x_1 \vee x_2) \wedge (x_3 \vee x_4)) \vee ((x_5 \vee x_6) \wedge (x_7 \vee x_8))$$

by implementing disjunctions using our compiler and conjunctions by running the two protocols in parallel.

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 $(x_1 \in \mathcal{L}_1) \vee (x_2 \in \mathcal{L}_2) \vee \ldots \vee (x_\ell \in \mathcal{L}_\ell)$. By "$\Sigma$-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, working with any random NP complete language will add the cost of 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. In Section **??** (in the supplementary material), we calculate the exact complexity of our compiler when instantiated with the partially binding vector commitments based on Pederson commitments for composing KKW.

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

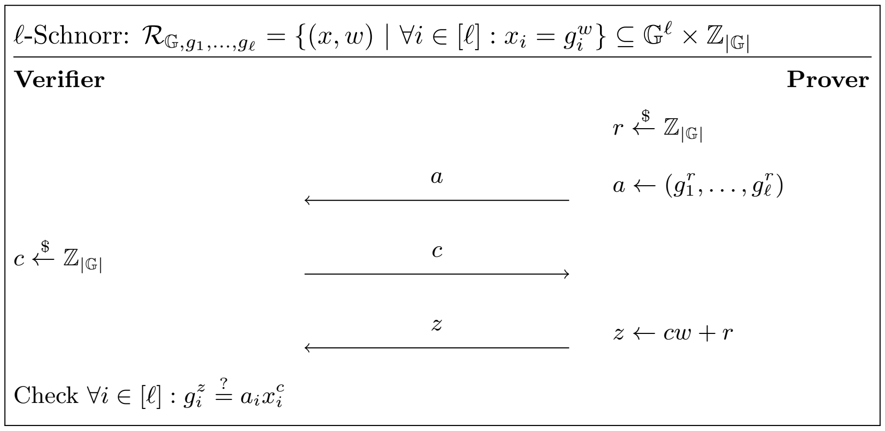
ℓ-Schnorr: $\mathcal{R}_{\mathbb{G},g_1,\dots,g_\ell} = \{(x,w) \mid \forall i \in [\ell] : x_i = g_i^w\} \subseteq \mathbb{G}^\ell \times \mathbb{Z}_{|\mathbb{G}|}$
Verifier
Prover
$r \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$
a
$a \leftarrow (g_1^r, \dots, g_\ell^r)$
$c \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$
c
z
$z \leftarrow cw + r$
Check $\forall i \in [\ell] : g_i^z \overset{?}{=} a_i x_i^c$

Figure 3: Generalized Schnorr

on different techniques. [12] 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-based protocols that doesn't require us to work with a meta-language:

Example 1 (ℓ-Schnorr) *Consider the naturally generalized ℓ-Schnorr protocol (included in the Supplementary Material in Figure 3) for the* NP *language with relation $\mathcal{R}_{\mathbb{G},g_1,\dots,g_\ell}(x = (x_1, \dots, x_\ell), w) : \forall i \in [\ell]\ x_i =^?= g_i^w$, where $\mathbb{G}$ is a cyclic group with generators $g_1, \dots, g_\ell$ and $w \in \mathbb{Z}_{|\mathbb{G}|}$. In this protocol, for all ℓ and choice of generators $\mathbb{G}, g_1, \dots, g_\ell$ and statement x, $\mathcal{D}_c^{(z)}$ is the uniform distribution over $\mathbb{Z}_{|\mathbb{G}|}$. Therefore, ℓ-Schnorr protocols can be stacked* as though *they were the same protocol using the self-stacking compiler when the group size $|\mathbb{G}|$ is the same.*

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 Rings) *Consider two KKW-based Σ-protocols. Σ_1 is for a language with a relation circuit defined over the ring $\mathbb{F}_{2^k}$, while Σ_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 depth of the relation circuits are the same, the the bit-wise distribution of $\mathcal{D}_c^{(z)}$ for Σ_1 and Σ_2 is the same (see Figure 11). Therefore, Σ_1 and Σ_2 can be stacked* as though *they were the same protocol using the self-stacking compiler and their*

[12] 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.

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 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 $\mathsf{TExt}_{\mathcal{D}\to\Sigma}$. For example, $F_{\Sigma\to\mathcal{D}}$ may add randomly sampled bits or cryptographic material to a third round message z in order to create an element $d \in \mathcal{D}$. $\mathsf{TExt}_{\mathcal{D}\to\Sigma}$ 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 10 (Cross Simulatability) *A stackable Σ-protocol $\Sigma = (A, Z, \phi)$ is 'cross simulatable' w.r.t. a distribution $\mathcal{D}$ if there exists a PPT algorithm $F_{\Sigma\to\mathcal{D}} : \mathcal{D}_c^{(z)} \to \mathcal{D}$ and a deterministic polynomial time algorithm $\mathsf{TExt}_{\mathcal{D}\to\Sigma} : \{0,1\}^\lambda \times \mathcal{D} \to \mathcal{D}_c^{(z)}$ satisfying the following properties:*

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

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

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

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

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

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

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 6. 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 Σ_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 Σ_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.*

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 4 (Cross-Stacking) *Let $\mathcal{D}$ be a distribution. For each $i \in [\ell]$, let $\Sigma_i = (A_i, Z_i, \phi_i)$ be a stackable (See Definition 7) 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* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Com}, \mathsf{Equiv})$ *be a 1-out-of-ℓ binding vector commitment scheme (See Definition 4). For any* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, the protocol $\Sigma' = (A', Z', \phi')$ described in Figure 4 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 the Appendix in Section F.

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 Σ' 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 $(|x_\mathcal{D}| + 2\log(\ell) O(\lambda) + |\mathsf{com}| + |r'|)$,

```
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 $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}; r)$ for $r \overset{\$}{\leftarrow} \{0,1\}^\lambda$.
    – Send $a = \mathsf{com}$ to Verifier.
– Second Round: Verifier samples $c \overset{\$}{\leftarrow} \{0,1\}^\lambda$ 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 $z_\alpha \leftarrow Z(x_\alpha, w_\alpha, c; r^p_\alpha)$.
    – $d \leftarrow F_{\Sigma_\alpha \to \mathcal{D}}(z_\alpha; r_{\text{map}})$
    – 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 $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', r)$
    – Send $z = (\mathsf{ck}, d, r')$ to the verifier.
– Verification: Verifier computes $\phi'(x, a, c, z) \to b$ as follows:
    – Parse $a = \mathsf{com}$ 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
          $b = \left(\mathsf{com} \overset{?}{=} \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}', r')\right) \wedge \left(\bigwedge_{i \in [\ell]} \phi(x_i, a_i, c, z_i)\right)$
    .
```

Figure 4: A compiler for stacking instances of multiple Σ-protocols.

where $O(\lambda)$ can be minimized by using efficient constructions of partially binding vector commitments, as shown in the complexity analysis of self-stacking.

Acknowledgments. 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. 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

[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. Cryptology ePrint Archive, Report 2020/753, 2020. `https://eprint.iacr.org/2020/753`.

[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.

[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.

[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.

[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`.

[CDG$^+$17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[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.

[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.

[DS16] Yuanxi Dai and John P. Steinberger. Indifferentiability of 8-round Feistel networks. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 95–120. Springer, Heidelberg, August 2016.

[FLPS20] Prastudy Fauzi, Helger Lipmaa, Zaira Pindado, and Janno Siim. Somewhere statistically binding commitment schemes with applications. Cryptology ePrint Archive, Report 2020/652, 2020. `https://eprint.iacr.org/2020/652`.

[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.

[HKP20a] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In *ASIACRYPT 2020, Part III*, LNCS, pages 3–30. Springer, Heidelberg, December 2020.

[HKP20b] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Motif: (almost) free branching in gmw via vector-scalar multiplication. 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.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.

[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.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

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

Appendix

A Constructions of Partially Binding Vector Commitments

In this section we explore constructions of partially binding vector commitment.

A.1 Random Oracles and Equivocal Commitments

In the random oracle model, partially binding commitments can be constructed from any equivocal commitment scheme wherein commitment keys ck are uniformly random bit strings. Recall the definition of a non-interactive equivocal commitment scheme:

Definition 11 (Non-interactive Equivocal Commitment) *A non-interactive equivocal commitment scheme* EC *with key space* $\mathcal{K}$*, message space* $\mathcal{M}$ *and randomness space* $\mathcal{R}$*, is defined by a tuple of the PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{TdGen}, \mathsf{Com}, \mathsf{Equiv})$ *defined as follows:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ *On input the security parameter* λ*, the setup algorithm outputs public parameters* pp*.*
- $\mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp})$*: On input the public parameters* pp*, the key generation algorithm outputs a commitment key* ck*.*
- $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TdGen}(\mathsf{pp})$*: On input the public parameter* pp*, the trapdoor key generation algorithm outputs a commitment key* ck *along with the corresponding trapdoor* td*.*
- $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, v; r)$*: On input the public parameter* pp*, the commitment key* ck*, trapdoor* td*, value* v *to be committed, this algorithm outputs the commitment* com*.*
- $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{td}, v, v', r)$*: On input the public parameter* pp*, the commitment key* ck*, the committed value* v*, the new value to be committed* v' *and opening randomness* r*, this algorithm outputs the new opening randomness* r'*.*

The properties satisfied by these algorithms are as follows:

1. ***Random Key Generation****: Let* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, then* $\mathsf{Gen}(\mathsf{pp})$ *is defined as* $\mathsf{ck} \overset{\$}{\leftarrow} \mathcal{K}$*.*
2. ***Oblivious Key Generation****: Let* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, then for all non-uniform PPT adversaries* $\mathcal{A}$*:*
$$\Pr[\mathcal{A}(\mathsf{ck}) = 1 \mid \mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp})] = \Pr[\mathcal{A}(\mathsf{ck}) = 1 \mid (\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TdGen}(\mathsf{pp})]$$
3. ***(Perfect) Hiding****: Let* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*,* $\mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp})$*, then for every pair* $v, v' \in \mathcal{M}$*:*
$$\{\mathsf{Com}(\mathsf{pp}, \mathsf{ck}, v; r) \mid r \overset{\$}{\leftarrow} \{0,1\}^\lambda\} \overset{p}{=} \{\mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \mathsf{td}, v'; r') \mid r' \overset{\$}{\leftarrow} \{0,1\}^\lambda\}$$
4. ***(Computational) Binding****: For any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *st.*
$$\begin{aligned}\mathsf{negl}(\lambda) \geq \Pr[v \neq v' \wedge \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, v; r) = \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, v'; r') \\ \mid (c, v, r, v', r') \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{ck}), \\ \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda), \mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp})]\end{aligned}$$
5. ***Equivocation****: For all* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*,* $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TdGen}(\mathsf{pp})$*,* $r \in \{0,1\}^\lambda$*,* $v, v' \in \mathcal{M}$*,* $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{td}, v, v', r)$ *equivocation succeeds:* $\mathsf{Com}(\mathsf{ck}, v; r) = \mathsf{Com}(\mathsf{ck}, v'; r')$

Lemma 7 (Partially Binding Vector Commitments From Equivocal Commitments) . *Let $\{P_i\}_{i\in[\ell]}$ distinct permutations sampled from a family of cryptographic permutations / wide random functions – in practice P_i be instanciated with a sufficiently large (tweaked) permutation, e.g. a variant of Keccack or Gimli. When the equivocal commitment scheme* EC *has a uniformly random distribution (over bits) of the commitment keys* ck*. The construction shown in Figure 5 is a t-out-of-ℓ binding vector commitment.*

Proof 4 *For $i \in [\ell]$ we model P_i as a separate invertible random oracle: these can be constructed from a single (non-invertible) random oracle using domain separation (e.g. 'append and hash') in a 8 round Feistel construction [DS16]. Completeness and hiding is clear from inspection (the scheme inherits the computation/perfect hiding of the equivocal commitment scheme* EC*). For partial binding:*

Assume there exists a PPT adversary $\mathcal{A}$ for which:

$$\Pr\left[D_{Ham}(\mathbf{v},\mathbf{v}') > \ell - t \;\wedge\; \mathsf{Com}(\mathsf{pp},\mathsf{ck},\mathbf{v};r) = \mathsf{Com}(\mathsf{pp},\mathsf{ck},\mathbf{v}';r') \;\middle|\; \begin{array}{c}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda);\\ (\mathsf{ck},\mathbf{v},\mathbf{v}',r,r') \leftarrow \mathcal{A}(1^\lambda,\mathsf{pp})\end{array}\right] \geq \frac{1}{p(\lambda)}$$

For some polynomial $p(\lambda)$. i.e. with non-negligible probability $\mathcal{A}$ returns $\mathsf{ck}, r, r', \mathbf{v}, \mathbf{v}'$ st. (1) $\mathsf{ck} = f(y) \in \mathbb{F}[y]$ with $deg(f) \leq \ell - t - 1$. (2) $r, r' \in \mathcal{R}^\ell$. (3) $\mathbf{v}, \mathbf{v}' \in \mathcal{M}^\ell$. Reduce to binding of the equivocal commitment scheme EC *as follows:*

1. *Get $\mathsf{ck}_i \xleftarrow{\$} \mathsf{Gen}(\mathsf{pp})$ from the computational binding game of* EC.
2. *Pick a random position $i \in [\ell]$*
3. *Pick a random query x to P_i on which $x = P_i^{-1}(\cdot)$ has not been queried previously and program $P_i(x) = \mathsf{ck}_i$. If $\mathcal{A}$ makes no query to P_i before returning $f(y)$ then program $P_i(f(i)) = \mathsf{ck}_i$.*
4. *Return $(\mathbf{v}_i, \mathbf{v}'_i, r_i, r'_i)$ as the response in the computational binding game of* EC.

To compute the probability of success: assume for contraction that there exists a $\ell - t + 1$ subset of indices $J \subseteq_{\ell-t+1} [\ell]$ with $\forall j \in J : P_j^{-1}(x_j) = f(j)$ for some $\{x_j\}_{j\in J}$ and $\mathcal{A}$ queries $P_j^{-1}(x_j)$ before $P_j(f(j))$. Let $J' \subsetneq_{\ell-t} J$ be a $\ell - t$ subset of J, observe that $\{f(j)\}_{j\in J'}$ fully determines $f(y)$, hence the probability that $P_j^{-1}(x_j) = f(j)$ for $j \in J \setminus J'$ is $1/|\mathbb{F}|$ which is negligible i.e. $\mathcal{A}$ cannot win the game with probability $1/p(\lambda)$, $\Rightarrow\!\Leftarrow$ Hence $\mathcal{A}$ can query $P_j^{-1}(x_j)$ before $P_j(f(j))$ in at most $\ell - t$ positions out of ℓ. Therefore: (1) The probability that we select such a position i is at most $(\ell-t)/\ell$ (2) The probability that we program the correct query $P_i(f(i))$ is at least $1/Q_i$ where Q_i is the number of queries to P_i, (3) The probability that $\mathbf{v}_i \neq \mathbf{v'}_i$ is at least $(D_{Ham}(\mathbf{v},\mathbf{v}') - \ell - t)/\ell \geq 1/\ell$. Hence the probability of winning the EC *binding game is at least: $\frac{\ell-t}{p(\lambda)\cdot Q_j \cdot \ell^2}$; which is non-negligible.*

Remark 2 *For the special case of $\ell = 2, t = 1$ (which is of primary interest when used in recursive stacking), the constraint introduced by f in the above construction can be expressed as $\mathsf{ck}_2 = P'(\mathsf{ck}_1)$ where $P' = P_2 \circ P_1^{-1}$ i.e. the commitment key $\mathsf{ck} = (\mathsf{ck}_1, \mathsf{ck}_2)$ can be represented as ck_1 from which ck_2 can be derived with a single evaluation of a suitable permutation P' (e.g. Keccak). Hence this is highly efficient in practice.*

A.1.1 From Pedersen Commitments (Discrete Log)

Given a family of cyclic groups $\mathbb{G}$ wherein discrete log (DL) is intractable EC can be instanciated with Pedersen commitments assuming elements of $\mathbb{G}$ can be encoded/decoded as uniformly random bit strings.

t-out-of-ℓ Binding Vector Commitment

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$: Output $\mathsf{pp} \leftarrow \mathsf{EC.Setup}(1^\lambda)$.
- $(\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B)$:
  - For each $i \notin B$, compute $(\mathsf{ck}_i, \mathsf{td}_i) \leftarrow \mathsf{EC.TdGen}(1^\lambda)$. For each $i \in B$, set $\mathsf{td}_i = \bot$.
  - Interpolate the degree $\ell - t - 1$ polynomial $f(y)$ st. $\forall i \notin B : f(i) = P_i^{-1}(\mathsf{ck}_i)$.
  - Output $\mathsf{ck} = f$ and $\mathsf{ek} = \{(i, \mathsf{td}_i)\}_{i \notin B}$.
- $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{v}; r)$
  - Parse $f(y) = \mathsf{ck}$, check $\deg(f) \leq \ell - t - 1$ otherwise output $\bot$.
  - Compute $\forall i \in [\ell] : \mathsf{ck}_i = P(f(i)); \mathsf{com}_i \leftarrow \mathsf{EC.Com}(\mathbf{v}_i; r_i)$
  - Output $\mathsf{com} = (\mathsf{com}_1, \ldots, \mathsf{com}_\ell)$
- $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \mathbf{v}, \mathbf{v}', r)$:
  - Compute $\forall i \in [\ell] \setminus B : r'_i \leftarrow \mathsf{EC.Equiv}(\mathsf{pp}, \mathsf{td}_i, \mathbf{v}_i, \mathbf{v}'_i, r_i)$
  - Let $\forall i \in B : r'_i \leftarrow r_i$.
  - Output $r' = (r'_1, \ldots, r'_\ell)$

Figure 5: Partially binding vector commitments from equivocal commitments and random oracles.

Non-interactive Equivocal Commitment from Dlog

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$. Generate a cyclic group $\mathbb{G}$ of order $|\mathbb{G}| > 2^\lambda$, sample $g \in \mathbb{G}$, output $\mathsf{pp} = (\mathbb{G}, g)$.
- $\mathsf{ck} \leftarrow \mathsf{Gen}(\mathsf{pp})$: Let $\mathsf{ck} \xleftarrow{\$} \mathbb{G}$.
- $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TdGen}(\mathsf{pp})$: Output $\mathsf{td} \leftarrow \mathbb{Z}_{|\mathbb{G}|}$, $\mathsf{ck} \leftarrow g^{\mathsf{td}}$.
- $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, v; r)$: Output $\mathsf{com} \leftarrow g^r \mathsf{ck}^v$
- $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{td}, c, v, v', r)$: Output $r' \leftarrow r + \mathsf{td} \cdot (v' - v)$

Figure 6: Pedersen commitments.

A.2 From Σ-Protocols (Interactive)

The partially binding commitment construction of [CPS+16] enables stacking in the plain model without increasing round complexity. The scheme has computation binding/hiding. We describe the construction here for completeness and cover its interaction with stacking; most notably the use of the EHVZK simulator which is needed for using the scheme with recursive stacking. Given a cyclic group $\mathbb{G}$ start by defining the two languages:

$$\mathcal{L}_{\mathrm{DH}} = \{(g_1, g_2, g_1^w, g_2^w)\}_{w \in \mathbb{Z}_{|\mathbb{G}|}} \subseteq \mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G}$$
$$\mathcal{L}_{\mathrm{DH1}} = \{(g_1, g_2, g_1^w, g_1 g_2^w\}_{w \in \mathbb{Z}_{|\mathbb{G}|}} \subseteq \mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\Sigma_{\mathrm{DH}}\text{: } \mathcal{R}_{\mathbb{G}} = \{(x = (g_1, g_2, h_1, h_2), w) \mid h_1 = g_1^w \wedge h_2 = g_2^w\} \subseteq (\mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G}) \times \mathbb{Z}_{|\mathbb{G}|}} \\ \hline
\textbf{Verifier} & & \textbf{Prover} \\
 & & r \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|} \\
 & \xleftarrow{\qquad a \qquad} & a = (g_1^r, g_2^r) \\
c \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|} & \xrightarrow{\qquad c \qquad} & \\
 & \xleftarrow{\qquad z \qquad} & z = cw + r \\
\text{Check } g_1^z = h_1^c a_1 \wedge g_2^z = h_2^c a_2 & &
\end{array}$$

Figure 7: Classical Σ-Protocol for the DH-tuple relation.

Clearly the languages are disjoint: $\mathcal{L}_{\mathrm{DH}} \cap \mathcal{L}_{\mathrm{DH1}} = \emptyset$. Under the Decisional Diffie-Hellman (DDH) assumption on $\mathbb{G}$ the uniform distribution over the two languages are computationally indistinguishable $\{s \mid s \xleftarrow{\$} \mathcal{L}_{\mathrm{DH}}\} \overset{c}{\approx} \{s \mid s \xleftarrow{\$} \mathcal{L}_{\mathrm{DH1}}\}$. A well-known standard Schnorr-style Σ-protocol (recalled in Figure 7) enables proving membership of $\mathcal{L}_{\mathrm{DH}}$. Membership of $\mathcal{L}_{\mathrm{DH1}}$ can be proved using a reduction to $\mathcal{L}_{\mathrm{DH}}$: $(g_1, g_2, h_1, h_2) \in \mathcal{L}_{\mathrm{DH}} \iff (g_1, g_2, h_1, g_1^{-1}h_2) \in \mathcal{L}_{\mathrm{DH1}}$. The Σ-protocol Σ_{DH1} is defined as:

- $A_{\mathrm{DH1}}(x = (g_1, g_2, h_1, h_2), w; r) \coloneqq A_{\mathrm{DH}}(x = (g_1, g_2, h_1, g_1^{-1}h_2), w; r)$
- $Z_{\mathrm{DH1}}(x = (g_1, g_2, h_1, h_2), w, c; r) \coloneqq Z_{\mathrm{DH}}(x = (g_1, g_2, h_1, g_1^{-1}h_2), w, c; r)$

The partially binding commitment scheme (covered in detail below) works by having the prover generate $\ell - t$ witness/instance pairs of $\mathcal{L}_{\mathrm{DH1}}$ and t witness/instance pairs of $\mathcal{L}_{\mathrm{DH}}$, then proving that he did so correctly. To commit, the prover treats the message vector $\mathbf{v}$ as a list of challenges for Σ_{DH} and:

1. Runs the simulator for Σ_{DH} using the message as the challenge for every position i where the instance x_i belongs in $\mathcal{L}_{\mathrm{DH1}}$
2. Generates the first round message honestly for every position i where the instance x_i belongs to $\mathcal{L}_{\mathrm{DH}}$.

The prover can equivocate on any position where the instance belongs in $\mathcal{L}_{\mathrm{DH}}$ by completing the Σ-protocol transcript for any challenge/message using his witness for $x_i \in \mathcal{L}_{\mathrm{DH}}$. Since at least $\ell - t$ of the statements belong in $\mathcal{L}_{\mathrm{DH1}}$ he cannot do so in the remaining positions: generating a second valid transcript would allow extraction of a witness by special soundness, however the statement is not in the language.

Generation (Interactive). Generation of the commitment key ck is interactive (2 moves). To create a commitment key ck and equivocation key ek for the partially binding vector commitment scheme with t binding positions B, start by sampling:

- t witness/instance pairs $\{(x_i, w_i)\}_{i \in B}$ for the $\mathcal{L}_{\mathrm{DH1}}$ language.
- $\ell - t$ witness/instance pairs $\{(x_i, w_i)\}_{i \in [\ell] \setminus B}$ for the $\mathcal{L}_{\mathrm{DH}}$ language.

Using the classical proof of partial knowledge compiler of [CDS94] on Σ_{DH1} the committer proves that at least t of $x_1, \ldots, x_\ell$ are in the language $\mathcal{L}_{\mathrm{DH1}}$ i.e. $\exists B \subseteq_t [\ell] : \forall i \in B : x_i \in \mathcal{L}_{\mathrm{DH1}}$. Let $\Sigma_{\mathrm{DH1},t\text{-of-}\ell}$ be the compiled Σ-protocol. Let $a_{\mathrm{DH1},t\text{-of-}\ell} \leftarrow A_{\mathrm{DH1},t\text{-of-}\ell}(x = (x_1, \ldots, x_\ell), w = (B, \{w_i\}_{i \in B}); r)$ be the first message of $\Sigma_{\mathrm{DH1},t\text{-of-}\ell}$ generated using the witness $w = (B, \{w_i\}_{i \in B})$. The prover sends $(x_1, \ldots, x_\ell, a_{\mathrm{DH1},t\text{-of-}\ell})$ to the verifier. The verifier then sends a random challenge $c \in \{0,1\}^\lambda$ in response, the prover (ck generator) computes $z_{\mathrm{DH1},t\text{-of-}\ell} \leftarrow Z_{\mathrm{DH1},t\text{-of-}\ell}(x = (x_1, \ldots, x_\ell), w = (B, \{w_i\}_{i \in B}), c; r)$. The commitment key is: $\mathsf{ck} = (x_1, \ldots, x_\ell, a_{\mathrm{DH1},t\text{-of-}\ell}, z_{\mathrm{DH1},t\text{-of-}\ell})$. The equivocation key is $\mathsf{ek} = \{w_i\}_{i \in [\ell] \setminus B}$.

Committing with trapdoor. To commit to $\mathbf{v} = (c_1, \ldots, c_\ell) \in (\{0,1\}^\lambda)^\ell$ with equivocation key $\mathsf{ek} = \{w_i\}_{i \in [\ell] \setminus B}$: (1) Sample $\forall i \in [\ell] \setminus B : z_i \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$. (2) Run the EHVZK simulator $\mathcal{S}_{\mathrm{DH}}^{\mathrm{EHVZK}}$ of Σ_{DH} for all the binding positions $\forall i \in B : a_i \leftarrow \mathcal{S}_{\mathrm{DH}}^{\mathrm{EHVZK}}(x_i, c_i, z_i)$. (3) Using the valid witness $\{w_i\}_{i \in [\ell] \setminus B}$ for $\forall i \in [\ell] \setminus B : x_i \in \mathcal{L}_{\mathrm{DH}}$ generate first round messages for the equivocal positions: for every $i \in [\ell] \setminus B$: sample randomness $t_i \xleftarrow{\$} \{0,1\}^\lambda$ and compute the first round message $a_i \leftarrow A_{\mathrm{DH}}(x_i, w_i; t_i)$. (4) Output the first-round messages $\mathsf{com} = (\mathsf{ck}, a_1, \ldots, a_\ell)$ as the commitment.

Committing without trapdoor. To commit to $\mathbf{v} = (c_1, \ldots, c_\ell) \in (\{0,1\}^\lambda)^\ell$ with randomness $r = (z_1, \ldots, z_\ell) \in \mathbb{Z}_{|\mathbb{G}|}^\ell$ run the EHVZK simulator $\mathcal{S}_{\mathrm{DH}}^{\mathrm{EHVZK}}$ of Σ_{DH} for all positions: $\forall i \in [\ell] : a_i \leftarrow \mathcal{S}_{\mathrm{DH}}^{\mathrm{EHVZK}}(x_i, c_i, z_i)$. Output $\mathsf{com} = (a_1, \ldots, a_\ell)$ as the commitment.

Equivocating. Given $\mathbf{v}, \mathbf{v}', \mathsf{ek} = \{w_i\}_{i \in [\ell] \setminus B}$ and $r = (z_1, \ldots, z_\ell)$. Compute $r' \in \mathbb{Z}_{|\mathbb{G}|}^\ell$ as follows:

$$\begin{aligned}
\forall i \in B &: \text{Copy the last-round message over } z_i' \leftarrow z_i \\
\forall i \in [\ell] \setminus B &: \text{Finish the } \Sigma\text{-protocol transript using the witness } z_i' \leftarrow Z_{\mathrm{DH}}(x_1, w_i, c_i = \mathbf{v}_i'; t_i)
\end{aligned}$$

Output $r' = (z_1', \ldots, z_\ell')$.

Observation about round complexity: If one naively has the prover/verifier generate commitment and equivocation keys $(\mathsf{ck}, \mathsf{ek})$ before running the compiled Σ-protocol (see Theorem 3), then the resulting protocol would have $2 + 3$ rounds: in particular the compiled protocol would not be a Σ-protocol. However (as done in the work by Ciampi *et. al.* [CPS+16]) interaction can be reduced to a 3 rounds: the additional round is introduced by the challenge to $\Sigma_{\mathrm{DH1},t\text{-of-}\ell}$ which ensures that the $x_1, \ldots, x_\ell$ instances were correctly generated, since ck is only sent in the third round in the compiled protocol, the verifier can send a challenge for the original Σ-protocol and for $\Sigma_{\mathrm{DH1},t\text{-of-}\ell}$ simultaneously in the second round.

A.3 Comparing Overheads For Choices Of Commitment Schemes

Below we briefly outline the overhead incurred by the partially binding commitment scheme for different instantiations and ℓ clauses:

- **DDH + RO**: Compiling Pedersen commitments using Lemma 7, we get a partially binding commitment scheme with $|\mathsf{ck}| = (\ell - 1)\lambda$ and $c \in \mathbb{G}$ a single group element and opening $r \in \mathbb{Z}_{|\mathbb{G}|}$. This translates to a overhead of $\lambda\ell$ bits [13] for ℓ positions. Or $\lambda 2 \log_2(\ell) + \mathtt{CC}(\Sigma) = 512 \cdot \log_2(\ell) + \mathtt{CC}(\Sigma)$ bits of total communication when recursively stacking ℓ clauses by instantiating $\mathbb{G}$ with a suitable 256-bit Elliptic curve offering 128-bits of security. This instantiation offers the best concrete performance.

- **Σ, DDH (with interaction)**: Using the construction from Ciampi *et al.* [CPS+16], gives an *interactive* partially binding commitments from DDH without random oracles with communication:

  $$\ell(2\log_2(|\mathbb{G}|) + \mathtt{CC}(\Sigma_{\mathrm{DDH1}}) + \mathtt{CC}(\Sigma_{\mathrm{DH}})) = \ell(2\log_2(|\mathbb{G}|) + 3\log_2(|\mathbb{G}|) + 3\log_2(|\mathbb{G}|)) = \ell 8\log_2(|\mathbb{G}|) = 8\ell\lambda$$

  In practice $\lambda = 256$, when using a suitable Elliptic curve. In which case the overhead is $\ell \cdot 2048$ bits.

B Well-Behaved Simulators: Proof of Lemma 6

Proof 5 *Given $\Sigma = (A, Z, \phi)$, construct the new $\Sigma' = (A', Z', \phi')$ 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 $z \leftarrow Z(x, w, c; r)$*

[13] In practice $\lambda = 256$ for Elliptic curves at $\lambda/2$-bits of security.

- $\phi'(x, a', c, z) :=$
  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 Σ' 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 Σ' first runs the simulator $\mathcal{S}$ of Σ, if the simulated transcript (a, c, z) is accepting then output the transcript, otherwise $\mathcal{S}'$ 'guesses' the challenge:

- $\mathcal{S}(1^\lambda, x, c) :=$
  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* $a' = (\perp, c)$*,* $z' = \perp$

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

- *Completeness: follows from completeness of Σ. 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 Σ. 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 assumpting that $a' = (\perp, c)$ is a contraction $\Rightarrow\Leftarrow$.*
  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 Σ.*

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

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

Σ has recyclable third messages $\implies$ Σ' 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.*

Blum: $\mathcal{R}_n(x, w) := w \in \mathrm{S}_n$ is a Hamiltonian cycle in the graph $x \in \{0,1\}^{n \times n}$

Verifier		**Prover**
	κ times in parallel	
		$\pi \xleftarrow{\$} \mathrm{S}_n$
		$r_{1,1}, \dots, r_{n,n} \xleftarrow{\$} \{0,1\}^\lambda$
	$\xleftarrow{a = (C_{1,1}, \dots, C_{n,n})}$	$\forall i, j \in [n]$:
		$C_{i,j} \leftarrow \mathsf{Commit}(x_{\pi(i),\pi(j)}; r_{\pi(i),\pi(j)})$
$c \xleftarrow{\$} \{0,1\}$	$\xrightarrow{c}$	
		$\tau \leftarrow \pi \circ w$
		$R_0 \leftarrow (\pi, r_{1,1}, \dots, r_{n,n})$
	$\xleftarrow{R_c}$	$R_1 \leftarrow (\tau, r_{1,\tau(1)}, \dots, r_{n,\tau(n),C_{1,1},\dots,C_{n,n}})$
if $c = 0$ compute $\forall i, j \in [n]$:		
$C_{i,j} \leftarrow \mathsf{Commit}(x_{\pi(i),\pi(j)}; r_{\pi(i),\pi(j)})$		
if $c = 1$ compute $\forall i \in [n]$:		
$C_{i,\tau(i)} \leftarrow \mathsf{Commit}(1; r_{i,\tau(i)})$		
Check: $a \stackrel{?}{=} (C_{1,1}, \dots, C_{n,n})$		

Figure 8: 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.

C Proofs from Section 6.2

In this section, we prove the lemmas from Section 6.2.

Guillou-Quisquater's Σ-protocol [GQ90]. Recall their for the pre-image relation $\mathcal{R}_f(x, w) := x \stackrel{?}{=} f(w)$ of a one-way group homomorphism f: (1) $A(x, w; r^p) := (f(r_1), \dots, f(r_\kappa))$ where $(r_1, \dots, r_\kappa) \xleftarrow{\$} \mathtt{dom}(f)^\kappa$. (2) $C(; r^v) := c$ where $c \xleftarrow{\$} \{0,1\}^\kappa$. (3) $Z(x, w, c; r^p) := (r_1 w^{c_1}, \dots, r_\kappa w^{c_\kappa})$. (4) $\phi(x, a, c, z)$: Check: $\forall i \in [\kappa] : f(z_i) \stackrel{?}{=} x^{c_i} a_i$.

Proof 6 (Lemma 2) *Let the extended simulator be $\mathcal{S}^{\text{EHVZK}}(1^\lambda, x, c, z) := (z_1^e x^{-c_1}, \dots, z_\kappa^e x^{-c_\kappa})$. Given statement $x \in \mathtt{range}(f)$ and challenge $c \in \{0,1\}^k$, let $\mathcal{D}_{x,c}^{(z)}$ be the uniform distribution over $\mathtt{range}(f)^\kappa$. Observe that the prover messages (a, z) in both real/simulated transcript are uniform subject to $\forall i \in [\kappa] : z_i^e = a_i x^{c_i}$. Hence this protocol is EHVZK. Moroever, since the above distribution $\mathcal{D}_{x,c}^{(z)}$ does not depend on x (or even c), this protocol also has recyclable third messages.*

Blum's Σ-Protocol [Blu87]. Let $\mathcal{L}_n^{\text{Ham}}$ be the language of n vertex graphs with a Hamiltonian cycle. We consider their protocol with a slight modifictation With a trivial modification: for any n Blum's Σ-protocol for $\mathcal{L}_n^{\text{Ham}}$ is stackable, recall the protocol (Shown in Figure 8):

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).

To be more precise:

Proof 7 (Lemma 3) *Define $\mathcal{D}_c^{(z)}$ as follows:*

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

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

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

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

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

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.

D Σ-Protocol of Katz, Kolesnikov and Wang

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 $\mathcal{R}(C, w) := C(w) = 1$ 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.

D.1 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 $f = (f_1, \dots, f_{|f|}) \in \{\texttt{Input}, \texttt{Add}, \texttt{Mul}, \texttt{Output}\}^{|f|}$ (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.

D.2 Underlaying 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 9) and sends the 'online players' $P_1, \dots, P_n$ 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 10) the n players

```
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 $\Delta$.
  • Send correlated randomness to each player: for $i \in [n]$ send $(\Delta, s_i)$ to $P_i$
```

Figure 9: KKW Preprocessing Player. R is any finite commutative ring.

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.

D.3 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 $\texttt{Input}(\gamma)$.

```
For every wire (with secret-shared value $v_\gamma$) 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 $(\Delta, s_i)$ 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 $z_\gamma$ on the broadcast channel.
        2. Regenerate the random sharing $[\lambda_\gamma]^{(i)} \overset{\$}{\leftarrow}_{s_i} \mathsf{R}$
           (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: $[\lambda_\gamma]^{(i)} \overset{\$}{\leftarrow}_{s_i} \mathsf{R}$
        2. Regenerate next Beaver share: $[\lambda_{\alpha,\beta}]^{(i)} \overset{\$}{\leftarrow}_{s_i} \mathsf{R}$
        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 $s_\gamma$ (broadcast $[s_\gamma]^{(i)}$)
        6. Locally compute $z_\gamma \leftarrow s_\gamma + z_\alpha z_\beta$
    – **if** $f_j = \texttt{Output}(\alpha)$:
        1. Reconstruct $\lambda_\alpha$ (broadcast $[\lambda_\alpha]^{(i)}$)
```

Figure 10: KKW Online Players. R is any finite commutative ring.

   (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 3 *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.

D.4 KKW is Stackable

We now prove that this MPC is $\mathcal{F}$-universally simulatable.

Proof 8 (Lemma 4) *Let $\mathcal{D}^{(real)}$ be the real distribution over condensed views for a particular $\mathcal{I}$. The simulator is given in Figure 11. Consider the two cases:*

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

- ***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 $\mathcal{H}^{(\Delta)}$:*

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

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

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

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

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

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

```
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: $\forall i \in \mathcal{I} : s_i \xleftarrow{\$} \{0,1\}^n$
Sample corrections: $\Delta \xleftarrow{\$} R^m$
return $(\mathcal{T}, \Delta, \{s_i\}_{i \in \mathcal{I}})$
```

Figure 11: Simulating the condensed views in KKW. R is the commutative ring over which the arithmetic circuits are computed.

E Overview of Ligero and Proof of Lemma 5

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.
- **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 1), 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 $\mathsf{ExpandViews}$ and $\mathsf{CondenseViews}$. But befor that, we borrow the following definitions from [AHIV17], which will aid in the description of the protocol.

Definition 12 (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 13 (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 14 (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 ith gate when evaluating $C(\alpha)$.
- It then constructs vectors x, y and z in $\mathbb{F}^{m\ell}$ where the jth entry of x, y and z contains the values β_a, β_b and β_c corresponding to the jth 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 jth row of $P_{\mathsf{add}} w$ equals $\beta_a + \beta_b - \beta_c$ where β_a, β_b and β_c correspond to the jth 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 jth 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 jth row of $P_{\mathsf{add}} w$ equals $\beta_a + \beta_b - \beta_c$ where β_a, β_b and β_c correspond to the jth addition gate in w.[14]

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

- 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:
  - $q^x[j] = u^x[j] + \Sigma_{i\in[m]} r_i^x(j) \cdot U^x[i,j] + \Sigma_{i=m+1}^{2m} r_i^x(j) \cdot U^w[i-m,j]$.
  - $q^y[j] = u^y[j] + \Sigma_{i\in[m]} r_i^y(j) \cdot U^y[i,j] + \Sigma_{i=m+1}^{2m} r_i^y(j) \cdot U^w[i-m,j]$.
  - $q^z[j] = u^z[j] + \Sigma_{i\in[m]} r_i^z(j) \cdot U^z[i,j] + \Sigma_{i=m+1}^{2m} r_i^z(j) \cdot U^w[i-m,j]$.
  - $p_0[j] = u^0[j] + \Sigma_{i\in[m]} r^q[i] \cdot (U^x[i,j] \cdot U^y[i,j] - U^z[i,j])$.

$\mathcal{F}$-universal simulator for Ligero MPC 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.
$\mathsf{ExpandViews}$: 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 $\{q^0[j]\}_{j\in[n]\setminus\mathcal{I}}$.

$\mathsf{CondenseViews}$: 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 Security Proof of Cross-Stacking Compiler (Theorem 4)

We now prove that the protocol $\Sigma' = (A', Z', \phi')$ described in Figure 4 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 Σ' using the extractors $\mathcal{E}_i$ for the underlying sigma protocols Σ_i. The extractor $\mathcal{E}'$ is given two accepting transcripts for the protocol Σ' 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 Σ_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-negligable probability, the $\mathcal{E}'$ succeeds with overwhelming probability.

Extended Honest-Verifier Zero-Knowledge (and Recyclable Third Messages). We denote distribution of third round message for Σ' 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 commitment key ck and randomness r:

```
${\mathcal{S}^{\text{EHVZK}}}'((x_1, \ldots, x_\ell), c, (\mathsf{ck}, r, d)) \coloneqq$
    $\forall i \in [\ell]$ compute $z_i \leftarrow \mathsf{TExt}_i(c, d)$
    $\forall i \in [\ell]$ compute $a_i \leftarrow \mathcal{S}_i^{\text{EHVZK}}(x_i, c, z_i)$
    **return** $a' = \mathsf{Com}(\mathsf{ck}, \mathbf{v} = (a_1, \ldots, a_\ell); r)$
```

Let $\mathcal{D}^{(\alpha,w)}$ denote the distribution of transcripts resulting from an honest prover possessing witness (α, w) running Σ' 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 $\mathcal{H}^{(\alpha,\mathsf{ck})} \overset{\text{p}}{=} \mathcal{H}^{(\alpha)}$ by the (perfect) hiding of the partially binding commitment scheme. Lastly note that $\mathcal{H}^{(\alpha,\mathsf{ck})}$ matches the output distribution of ${\mathcal{S}^{\text{EHVZK}}}'((x_1, \ldots, x_\ell), c, {\mathcal{D}_c^{(z)}}')$.

Therefore Σ' is a stackable Σ-protocol.

G k-out-of-ℓ Proofs Of Partial Knowledge

In this section, we show how our cross stacking compiler from the previous section can be extended to obtain efficient k-out-of-ℓ proofs of partial knowledge. These are proofs for statements of the form $x = x_1, \ldots, x_\ell$, where the prover must prove that it has witnesses to atleast k out of these ℓ clauses. More formally for languages of the form $\mathcal{L} = \{x = x_1, \ldots, x_\ell \mid \exists \mathbb{K} \subset [\ell], \{w_i\}_{i \in \mathbb{K}}, \{\mathcal{R}_i(x_i, w_1) = 1\}_{i \in \mathbb{K}}\}$.

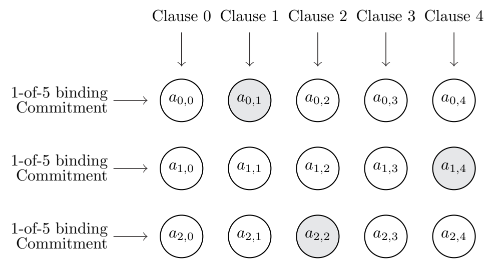

Figure 12: The structure of the commitment required for proving a 3-of-5. The grey circuits represent example binding positions for a valid proof. The third round message consists of 3 third round messages z_0, z_1, z_2. The verifier uses the first commitment to check the correctness of z_0, the second commitment to check the correctness of z_1, and the third commitment to check the correctness of z_3. Special care must be taken to ensure that no more than a single binding position appears in any given column.

A strawman approach towards designing a stacking compiler for such statements would be to run k independent instances of our cross stacking compiler, once for each of the k clauses for which the prover has a witness. However, this simple idea is vulnerable to an adversarial prover who might only have a witness for one of the clauses and uses that in each of the k instances. For proofs of partial knowledge, we need to make sure that the prover indeed has witnesses for k distinct clauses. For this, we extend the notion of partially binding vector commitments to accommodate partially binding commitments to matrices (or vectors-of-vectors), that allows a committer to commit to a matrix of $k \times \ell$ dimension such that only one entry in each row is binding. Moreover, the binding position in each of the rows must be distinct as shown in Figure 12. As a result, as shown in Figure 12, each row in these commitments is equivalent to a 1-out-of-ℓ binding vector commitment (See Definition 4).

We present a simple construction for this primitive using $k \times \ell$ instances of a regular non-interactive commitment and a non-interactive zero-knowledge proof.

G.1 Partially Equivocal Vector-of-Vectors Commitments

In this section, we formalize the notion of a k-out-of-ℓ binding vector-of-vectors commitment.

Definition 15 (k-out-of-ℓ Binding Vector-of-Vectors Commitment) *A k-out-of-ℓ binding non-interactive vector-of-vectors commitment scheme with message space $\mathcal{M}$, is defined by a tuple of the PPT algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Com}, \mathsf{Open}, \mathsf{Equiv})$ *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*. We denote the elements of B (in order) as b_i for $i \in [k]$.*
- $(\mathsf{com}, \mathsf{op}) \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \{\mathbf{v}_i\}_{i\in[k]}; r)$*: Takes public parameter* pp*, commitment key* ck*, and k ℓ-tuples $\mathbf{v}_i$ for $i \in [k]$ and randomness r. Returns a partially binding vector-of-vector commitment* com *along with an opening* op*. We denote the elements of the tuple $\mathbf{v}_i$ as $\mathbf{v}_{i,j}$ for $j \in [\ell]$*
- $b = \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}_i\}_{i\in[k]}, \mathsf{op})$*: On input the public parameter* pp*, commitment key* ck*, the commitment* com*, the committed values $\{\mathbf{v}_i\}_{i\in[k]}$ and the opening* op*, this algorithm outputs a bit $b \in \{0, 1\}$.*
- $\mathsf{op}' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}_i\}_{i\in[k]}, \{\mathbf{v}'_i\}_{i\in[k]}, \mathsf{com}, \mathsf{op})$*: Takes public parameters* pp*, equivocation key* ek*, commitment* com*, original opening* op*, original commitment value $\{\mathbf{v}_i\}_{i\in[k]}$, updated commitment values $\{\mathbf{v}'_i\}_{i\in[k]}$ such that $\forall i \in [k] : \mathbf{v}_{i,b_i} = \mathbf{v}'_{i,b_i}$. Returns an opening* op' *of the commitment $c = \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \{\mathbf{v}_i\}_{i\in[k]}; r)$ to the value $\{\mathbf{v}'_i\}_{i\in[k]}$*

The properties satisfied by these algorithms are as follows:

(Perfect) Hiding: *The commitment key* ck *(perfectly) hides the binding positions* B *and commitments* com *(perfectly) hide the committed* $k \times \ell$ *values in* $\mathbf{v}$*:*

$$\forall \{\mathbf{v}_i\}_{i\in[k]}, \{\mathbf{v}'_i\}_{i\in[k]} \in \mathcal{M}^{k\times\ell} : \forall B, B' \in \binom{[\ell]}{k} :$$

$$[(\mathsf{ck}, \mathsf{com}) | (\mathsf{com}, \mathsf{op}) \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \{\mathbf{v}_i\}_{i\in[k]}; r); r \leftarrow \{0,1\}^n; (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B); \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)] \overset{p}{=}$$
$$[(\mathsf{ck}', \mathsf{com}') | (\mathsf{com}, \mathsf{op}) \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}', \{\mathbf{v}'_i\}_{i\in[k]}; r); r \leftarrow \{0,1\}^n; (\mathsf{ck}', \mathsf{ek}') \leftarrow \mathsf{Gen}(\mathsf{pp}, B'); \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)]$$

(Computational) Partial Binding: *It is intractable to find a commitment key* ck*, commitment* com*.* ℓ*-tuples* $\{\mathbf{v}\}_{i\in[k]}, \{\mathbf{v}'\}_{i\in[k]}$ *and openings* $\mathsf{op}, \mathsf{op}'$ *such that, for all PPT algorithms* $\mathcal{A}$*:*

$$\Pr\left[\begin{array}{c|c} \forall B \in \binom{[\ell]}{k}, \exists i \in [k], \mathbf{v}_{i,b_i} \neq \mathbf{v}'_{i,b_i} \ \wedge & \\ \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}_i\}_{i\in[k]}, \mathsf{op}) = 1 \ \wedge & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); \\ \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}'_i\}_{i\in[k]}, \mathsf{op}') = 1 & (\mathsf{ck}, \mathsf{com}, \mathbf{v}, \mathbf{v}', \mathsf{op}, \mathsf{op}') \leftarrow \mathcal{A}(1^n, \mathsf{pp}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

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* $\forall b_i \in B : \mathbf{v}_{i,b_i} = \mathbf{v}'_{i,b_i}$*. That is,* $\forall\ B \in \binom{[\ell]}{k}$*,* $\forall\ \{\mathbf{v}_i\}_{i\in[k]}, \{\mathbf{v}'_i\}_{i\in[k]} \in \mathcal{M}^{k\times\ell}$ *st.* $\forall b_i \in B : \mathbf{v}_{i,b_i} = \mathbf{v}'_{i,b_i}$ *then:*

$$\Pr\left[\begin{array}{c|c} & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); \\ \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}'_i\}_{i\in[k]}, \mathsf{op}') = 1 & (\mathsf{ck}, \mathsf{ek}) \leftarrow \mathsf{Gen}(\mathsf{pp}, B); r \leftarrow \{0,1\}^n; \\ & (\mathsf{com}, \mathsf{open}) \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \{\mathbf{v}_i\}_{i\in[k]}; r) \\ & \mathsf{op}' \leftarrow \mathsf{Equiv}(\mathsf{ek}, \{\mathbf{v}_i\}_{i\in[k]}, \{\mathbf{v}'_i\}_{i\in[k]}, \mathsf{com}, \mathsf{op}) \end{array}\right] = 1$$

Construction in the Random Oracle Model. Let NICom be a non-interactive commitment scheme 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 13.

Theorem 5 *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 13 is a non-interactive* k*-out-of-*ℓ *binding vector-of-vectors commitment in the random oracle model.*

Proof 9 *Hiding follows from the hiding of the underlying commitment scheme. Binding and equivocation follow from the soundness and completementness of the zero-knowledge protocol.*

G.2 Stacking compiler

Theorem 6 (Stacking for Proofs of Partial Knowledge) *Let* $\mathcal{D}$ *be a distribution. For each* $i \in [\ell]$*, let* $\Sigma_i = (A_i, C_i, Z_i, \phi_i)$ *be a stackable (See Definition 7)* 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* $(\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Equiv})$ *be a* k*-out-of-*ℓ *binding vector-of-vectors commitment scheme (See Definition 15). For any* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, the protocol* $\Sigma' = (A', C', Z', \phi')$ *described in Figure 14 is a* Σ*-protocol for the relation* $\mathcal{R}'((x_1, \ldots, x_\ell), (\mathbb{K} = \{k_1, \ldots, k_k\}, \{w_j\}_{j\in\mathbb{K}})) := \wedge_{j\in\mathbb{K}} \mathcal{R}_j(x_j, w_j)$*.*

Proof 10 ***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.*

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 $(\mathsf{ck}, \mathsf{ek}) = \perp$
- $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{ck}, \{\mathbf{v}_i\}_{i\in[k]}; r)$:
  - For each $i \in [k], j \in [\ell]$, sample a random value $r_{i,j}$ and compute $\mathsf{com}_{i,j} = \mathsf{NICom}(\mathbf{v}_{i,j}; r_{i,j})$.
  - Use the non-interactive zero-knowledge proof to compute a proof π using statetement $\{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}, \mathbf{v}$ and witness $B, \{r_{i,b_i}\}_{i\in[k]}$, for the following language:

$$\mathcal{L} = \left\{ \big(\{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}, \mathbf{v}\big) : \exists B \in \binom{[\ell]}{k}, \exists \{r_{i,b_i}\}_{i\in[k]}, \text{ such that } \mathsf{com}_{i,j} = \mathsf{NICom}(\mathbf{v}_{i,j}; r_{i,j}) \right\}$$

  - Set $\mathsf{com} = \{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}$ and $r = \pi$.
- $r' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}_i\}_{i\in[k]}, \{\mathbf{v}'_i\}_{i\in[k]}, r)$: Use the non-interactive zero-knowledge proof to compute a proof π using statetement $\{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}, \mathbf{v}'$ and witness $B, \{r_{i,b_i}\}_{i\in[k]}$, for the above language $\mathcal{L}$ and output $r' = \pi$.
- $b \leftarrow \mathsf{Open}(\mathsf{pp}, \mathsf{ck}, \mathsf{com}, \{\mathbf{v}_i\}_{i\in[k]}, r)$: Parse $\mathsf{com} = \{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}$ and $r = \pi$. Output $b = \mathsf{V}(\{\mathsf{com}_{i,j}\}_{i\in[k],j\in[\ell]}, \mathbf{v}, \pi)$.

Figure 13: A k-out-of-ℓ Binding Vector-of-Vectors Commitment

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

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

```
– First Round: Prover computes $A'(x, w; r^p) \to a$ as follows:
    – Parse $r^p = (\{r_j^p\}_{j \in \mathbb{K}} \| r \| \{r_{\mathrm{map},j}\}_{j \in [k]})$.
    – 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 $\mathbf{v}_j = (v_{j,1}, \dots, v_{j,\ell})$
    – Compute $(\mathsf{com}, \mathsf{op}) \leftarrow \mathsf{Com}(\mathsf{pp}, \mathsf{ck}, \{\mathbf{v}_j\}_{j \in [k]}; r)$.
    – 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 $r^p = (\{r_j^p\}_{j \in \mathbb{K}} \| r \| \{r_{\mathrm{map},j}\}_{j \in [k]})$.
    – For each $j \in [k]$:
        * Compute $z_j \leftarrow Z_{k_j}(x_{k_j}, w_{k_j}, c; r_{k_j}^p)$
        * Compute $d_j \leftarrow F_{\Sigma_{k_j} \to \mathcal{D}}(z_j; r_{\mathrm{map},j})$
        * For $i \in [\ell] \setminus k_j$,
            · Set $z_{j,i} \leftarrow \mathsf{TExt}_i(c, d_j)$
            · Set $a_{j,i} \leftarrow \mathcal{S}_i^{\mathrm{EHVZK}}(x_i, c, z_{j,i})$
        * Set $a_{j,k_j} \leftarrow a_{k_j}$
    – For each $j \in [k]$, set $\mathbf{v}'_j = (a_{j,1}, \dots, a_{j,\ell})$
    – Compute $\mathsf{op}' \leftarrow \mathsf{Equiv}(\mathsf{pp}, \mathsf{ek}, \{\mathbf{v}_j\}_{j \in [k]}, \{\mathbf{v}'_j\}_{j \in [k]}, \mathsf{com}, \mathsf{op})$
    – Compute and send $z = (\mathsf{ck}, \{d_j\}_{j \in [k]}, \mathsf{op}')$ to the verifier.
– Verification: Verifier computes $\phi'(x, a, c, z) \to b$ as follows:
    – Parse $a = \mathsf{com}$ and $z = (\mathsf{ck}, \{d_j\}_{j \in [k]}, \mathsf{op}')$
    – 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 $a_{j,i} \leftarrow \mathcal{S}_i^{\mathrm{EHVZK}}(x_i, c, z_{j,i})$
    – For each $j \in [k]$, set $\mathbf{v}'_j = (a_{j,1}, \dots, a_{j,\ell})$
    – Compute and return $b$ as
```

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

Figure 14: A compiler for k-out-of-ℓ proofs of partial knowledge.