Either/Or

Mathias Hall-Andersen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Either/Or

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Mathias Hall-Andersen
January 12, 2024

Abstract

Many models of computation in computer science allow for conditional execution, for example RAM machines with instructions such as conditional jumps: where the address of the next instruction depends on the result of a comparison. This means that only part of the program is executed, depending on the result of the comparison. This is in contrast to e.g. Boolean circuits, where all operations (AND/XOR/OR etc.) are executed, regardless of the input. In the zero-knowledge and secure MPC literature, the primary focus has been on computations without conditional execution, namely functions represented as circuits over a finite field. This means that if the function is best represented in a model with conditional execution, it will typically result in all possible executions (branches) being executed, after which the correct result is chosen based on the input. This is inefficient, because the communication and prover/verifier complexity is proportional to the number of branches: in contrast to executing the function in a model with conditional execution, where only one branch (the active one) is executed.

This thesis covers techniques for proving branching computation / disjunctions inside zero-knowledge proofs, as well as related techniques for executing branching computation in multiparty computation. The following three works have been selected for the thesis: 1. *Stacking Sigmas: A Framework to Compose* Σ*-Protocols for Disjunctions*. Which proposed a concretely efficient compiler (Stacking Sigmas) to convert Σ-protocol into proofs of disjunctions with logarithmic overhead in the number of clauses. 2. *Curve Trees: Practical and Transparent Zero-Knowledge Accumulators*. Building concretely efficient cryptographic accumulator with efficient membership proofs (a disjunction over the members of a set). from commit-and-proof techniques and a novel use of elliptic curve cycles. 3. *Secure Multiparty Computation with Branching*. Constructs efficient protocols for executing conditional branching ("if-statements") inside multiparty computation.

Resumé

Mange beregningsmodeller ("models of computation") indenfor datalogi tillader betinget udførelse, for eksempel RAM machiner med maskininstruktioner såsom betingede hop ("conditional jumps"): hvor addresses af den næste instruktion afhænger af resultatet af en sammenligning. Derved bliver kun en del af programmet udført, afhængigt af resultatet af sammenligningen. Dette er i modsætning til f.eks. Boolske kredsløb, hvor alle operations (AND/XOR/OR etc.) bliver eksekveret, unanset inputtet. I zero-knowledge og sikker MPC ("Multiparty Computation") literaturen har det primære fokus været på beregninger som ikke har betinget udførelse, navnlig funktioner representeret som kredsløb over et endeligt legeme. Dette betyder at hvis funktionen er bedst representeret i en model med betinget udførelse, så vil det typisk resultere i at alle mulige udførelser ("branches") bliver eksekveret, hvorefter det korrekte result bliver valgt ud fra inputtet. Dette er ineffektivt, fordi kommunikationen og beviserens køretid er proportional med antallet af branches: i modsætning til eksekvering af funktionen i en model med betinget udførelse, hvor kun en "branch" (den aktive) bliver eksekveret.

Denne afhandling dækker teknikker til at bevise forgrening ("branching") / disjunktioner i zero-knowledge beviser, samt relaterede teknikker til at eksekvere forgrening i multiparty computation. De følgende tre værker er blevet udvalgt til afhandlingen: 1. *Stacking Sigmas: A Framework to Compose Σ-Protocols for Disjunctions*. Foreslog en konkret effektiv kompiler (Stacking Sigmas) til at konvertere Σ-protokoller til beviser for disjunktioner med logaritmisk overhead i antallet af klausuler. 2. *Curve Trees: Practical and Transparent Zero-Knowledge Accumulators*. Bygger konkret effektive kryptografiske akkumulatorer med effektive medlemskabsbeviser (en disjunktion over elementerne af en mængde) fra commit-and-proof teknikker og en ny anvendelse af elliptiske kurve cykler. 3. *Secure Multiparty Computation with Branching*. Konstruerer effektive protokoller til at eksekvere forgreninger ("if-statements") over et antal kredsløb i multiparty computation.

Acknowledgments

The whole of the cryptography group at Aarhus University, a truly unique professional and social environment filled with brilliant, kind and curious people. Among them, my advisor, Jesper Buus Nielsen, for convincing me to do a PhD in the first place and for being a great mentor throughout the process. Your enthusiasm for the whole of cryptography and your desire to see it applied is inspiring.

Thanks to all my co-authors. In no particular order: Gabe Kaptchuk, Aarushi Goel, Jesper Buus Nielsen, Nicholas Spooner, Abhishek Jain, Aditya Hegde, Matthew Green, Nikolaj Schwartzbach, Gijs Van Laer, Matteo Campanelli, Simon Holmgaard Kamp, Mark Simkin, Benedikt Wagner, Diego F. Aranha, Anca Nitulescu, Elena Pagnin, Sophia Yakoubov. It has been a pleasure working with you all.

I would like to thank the wider cryptographic community and the International Association of Cryptologic Research (IACR) in particular. In an era of academic rent-seeking, it is remarkable to be in a field that has ownership of its own community and is committed to openly sharing their work, rather than hiding it behind expensive subscriptions and charging exorbitant fees for publishing. As an undergraduate, showing up to conferences and trying to learn, I was welcomed into the community, encouraged to participate and learned that cryptographers are almost universally approachable, down-to-earth, willing to explain and excited about cryptography as a field. There are of course always things to improve, but it could also be so much worse...

I want to thank my family. My parents, for always supporting me and encouraging me to pursue my interests – even if not fully grasping them at all times. My brother, for being a great friend throughout my childhood.

And last, but not least, I want to thank my wonderful girlfriend, Juci. For keeping me sane in times of stress, for giving me a world outside of cryptography, for being my partner and being my best friend over the last couple of years. I will always remember my time at Aarhus University as the time I met you.

Mathias Hall-Andersen,
Copenhagen, January 12, 2024.

Contents

Part I

Overview

Chapter 1

Introduction to Cryptography

> "Anyone who considers arithmetical methods of
> producing random digits is, of course, in a state of sin."
> – John von Neumann (1951) on Pseudorandom Generators (PRGs) [Neu51]

1.1 Notation

We define NP relations by their polynomial time computable predicates $\mathscr{R}(x,w) \mapsto \{0,1\}$, rather than the more common definition of a relation as a subset of tuples. This is done for convenience, as it allows us to unify notation across the Multi-Party Computation and Zero-Knowledge sections. Throughout the thesis, λ denotes the "computational security parameter" and κ denotes the "statistical security parameter".

1.2 Cryptography in a Nutshell

Early Days. Cryptography, from the greek *kryptós* (hidden) and *gráphein* (to write), originated as the study of "secret writing", with what we today would classify as *symmetric-key encryption*. Most laypeople, when they think of cryptography, think of this type of encryption, whether it be the Caesar cipher or the Enigma machine. In this classical form, cryptography has likely been around as long as writing itself, however modern cryptography has its origins in the three decades of the 1970s, 1980s and 1990s spurred by the discovery of public key cryptography. Which formalized and expanded the field, most importantly, by introducing formal assumptions, security definitions and proving security reductions. Taking cryptography from a bespoke art of "seems to not be broken" to a rigorous science, this formalization of the field yielded an explosion of new cryptographic primitives and constructions. Today cryptography is perhaps best described as the constructive study of intractable problems: namely, what schemes/protocols can we build assuming the existence of certain computationally intractable problems or computationally indistinguishable/unlearnable distributions. As a field, cryptography is therefore the polar opposite of algorithmics / cryptanalysis

$\mathsf{Game}_{\mathsf{OWF}}(\lambda, \mathscr{A})$

/ Game picks a random preimage

1 : $x \xleftarrow{\$} \{0,1\}^\lambda$

2 : $y \leftarrow f_\lambda(x)$

/ Ask the adversary to invert the function

3 : $x' \leftarrow \mathscr{A}(1^\lambda, y)$

/ Adversary wins if he finds any preimage

4 : **if** $f(x') = y$: **return** 1

5 : **else return** 0

Figure 1.1: Inversion game for One-Way Functions

which aims to reduce the complexity of solving problems and of machine learning / learning theory which aims to identify and learn distributions.

Cryptographic Assumptions and Games. Any non-trivial cryptography requires, *in particular* [1] that $\mathsf{P} \neq \mathsf{NP}$, famously an open problem in computer science, therefore, cryptography crucially relies on unpoven conjectures in complexity theory. Cryptography can only get off the ground if we *assume* the *hardness of NP languages* with respect to some distribution over instances, or, even the polynomial-time indistinguishability between distributions over two languages. Any such *cryptographic assumptions* can be captured by the notion of a game played by a polynomially bounded adversary $\mathscr{A}$ (a PPT Turing machine), the *assumption* is then simply a bound on which *any* such adversary $\mathscr{A}$ could win the game. The most basic cryptographic assumption is the *existence of one-way functions* (OWF), which is captured by the assumption below:

Assumption 1 (Existance Of One-Way Functions.) *There exists a family of polynomial-time computable functions $\mathscr{F} = \{f_\lambda : \{0,1\}^\lambda \to \{0,1\}^\lambda\}$, such that for all PPT adversaries $\mathscr{A}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that the probability that $\mathscr{A}$ wins the game* $\mathsf{Game}_{\mathsf{OWF}}$ *in Figure 1.1 is negligible:*

$$\forall \mathscr{A} \in \mathsf{PPT}. \exists \mathsf{negl}(\lambda). \Pr[\mathsf{Game}_{\mathsf{OWF}}(\lambda, \mathscr{A}) = 1] \leq \mathsf{negl}(\lambda)$$

One-way functions are about as simple as cryptographic assumptions get, but even in this case note a proof of Assumption 1 would trivially imply $\mathsf{P} \neq \mathsf{NP}$, let:

$$\mathscr{L} = \{(f_\lambda(x), i, x_{1,\ldots,i}) \mid x \in \{0,1\}^\lambda, y \in \{0,1\}^*\}$$

i.e. tuples of images, for which there exists a preimage x with a given prefix $x_{1,\ldots,i}$ of length i. Then it is clear to see if $\mathscr{L} \in \mathsf{P}$ the adversary $\mathscr{A}$ can use the decider to do

[1] A necessary, but not sufficient condition.

$\mathsf{Game}_{\text{IND-CPA}}(\lambda, \mathscr{A})$

// Adversary picks two messages of equal length
1 : $(\mathsf{pt}_0, \mathsf{pt}_1) \leftarrow \mathscr{A}(1^\lambda)$
2 : **if** $|\mathsf{pt}_0| \neq |\mathsf{pt}_1|$ **return** 0
// Game encrypts one of the messages at random
3 : $\mathsf{k} \xleftarrow{\$} \{0,1\}^\lambda$
4 : $b \xleftarrow{\$} \{0,1\}$
5 : $\mathsf{ct} \leftarrow \mathsf{Enc}_\mathsf{k}(\mathsf{pt}_b)$
// Adversary tries to guess which message was encrypted
6 : $b' \leftarrow \mathscr{A}(\mathsf{ct})$
7 : **if** $b' = b$ **return** 1
8 : **return** 0

Figure 1.2: IND-CPA security game for a symmetric encryption scheme Enc

a binary search for the next bit x_i of some preimage x. It is also obvious that x is a certificate for the NP relation, hence if Assumption 1 is true, then $\mathscr{L} \in \mathsf{NP} \setminus \mathsf{P}$.

What Is Reasonable To Assume? It might seem strange to other mathematicians and computer scientists to have an entire field which is entirely predicated on numerous (strong) unproven and sometimes *very specific* conjectures in complexity theory: for instance the conjectured hardness of certain problems in algebraic geometry. So what are we "allowed" to assume? If we can just define a game for our constructions and *assume away the existence* of any polynomial time Turing machines which may win it, does cryptography become a vacuous endeavour? The answer is that it depends on how *weak/plausible* the assumptions are. For instance, the existence of one-way functions defined above is much more plausible, than the security of the whole of the Transport Layer Security (TLS) protocol. In cryptography, the set of assumptions that we are willing to make forms the set of *axioms* from which we can prove statements (security of protocols). By expanding the set of cryptographic assumptions we are willing to make, we can prove more and more protocols secure, in exactly the same way that mathematicians can prove more and more theorems by expanding the set of axioms they are willing to include, e.g. choosing whether to include Choice in Zermelo-Fraenkel set theory. Making an additional cryptographic assumption, is then equivalent to adding a theorem to the axiom set. Cryptographers, like mathematicians, are therefore interested in proving security from as few/as natural axioms as possible: meaning making as few and as weak/strong assumptions as possible.

Indistinguishablity as a Security Definition. A special case of games is *indistinguishability* games, in which the adversary $\mathscr{A}$, called a distinguisher (sometimes denoted as Dist) in this context, is given samples from one of two distributions Game_0 and Game_1, and must guess which distribution the samples are from better than ran-

dom guessing. Formally, the game flips a coin $b \xleftarrow{\$} \{0,1\}$, runs Game_b with $\mathscr{A}$. The adversary $\mathscr{A}$ then outputs a guess b', and wins if $b = b'$. We say that two distributions Game_0 and Game_1 are indistinguishable if no adversary $\mathscr{A}$ can win the indistinguishability game with probability better than $\frac{1}{2} + \varepsilon$. Various flavors of indistinguishability are widely used:

Perfect Indistinguishability: In which $\varepsilon = 0$ and the adversary $\mathscr{A}$ is a computationally unbounded Turing machine.

Statistical Indistinguishability: In which $\varepsilon < 2^{-\kappa}$ and the adversary $\mathscr{A}$ is a computationally unbounded Turing machine.

Computational Indistinguishability: In which $\varepsilon = \mathsf{negl}(\lambda)$, for a negligible function $\mathsf{negl}(\lambda)$ and the adversary $\mathscr{A}$ is a probabilistic polynomial time (in 1^λ) Turing machine.

For notational convenience, we will use $\mathsf{Game}_0 \approx \mathsf{Game}_1$ to indistinguishability between Game_0 and Game_1, with the type of indistinguishability being implicit: essentially all definitions in cryptography come in a perfect, statistical and computational flavor, depending on the classes of adversary $\mathscr{A}$ that is considered and his success probability ε.

This general framework of indistinguishability games is a convenient way to define security of cryptographic schemes and protocols for numerous reasons:

It is simple: It is easy to define and reason about, e.g. rather than trying to define what it means for a scheme to be 'secure', we simply require it to be indistinguishable from some trivially secure ideal scheme.

It is strong: Any reasonable definition of a 'break' for some scheme *would imply* the existance of a distinguisher. e.g. if a symmetric encryption scheme is 'broken' in the sense that an adversary can decrypt a ciphertext without knowing the key, for a set of weak keys with density ε, then we can trivially construct a distinguisher that wins the IND-CPA indistinguishability game with probability ε.

It composes: By relying on indistinguishability, we can "chain together" hybrid distributions in which we switch parts of a larger protocol from one of the two distributions to the other. For instance, in a security proof of a protocol, we might create a hybrid in which we switch commitments to 'real' values with commitments to random values, and then use indistinguishability of the commitment to argue that the adversary cannot tell the difference between the real protocol and this hybrid. In subsequent hybrids we might switch other parts of the protocol, step-by-step, until we have arrived at the ideal world.

The universal use of indistinguishability as the basis of security definitions is what has enabled us to build and prove security of complex protocols from simple building

blocks. Now that we have an idea of what the axioms/assumptions/theorems/games are, we can start to define what a proof is in this context.

Security Proofs as Karp Reductions. By defining both assumptions and security definitions as games, we can define security proofs as reductions between games: if the $\mathscr{A}$ succeeds in $\mathsf{Game}^{(\mathsf{Primitive})}$ with probability ε, then we can construct an $\mathscr{A}'$ that succeeds in $\mathsf{Game}^{(\mathsf{Assumption})}$ with probability ε'. The form that these reductions take are essentially Karp reductions, except that the reduction is interactive, may be randomized and may rely on rewinding.

Running Time of Games. In the definition of games, we have not specified the running time of the game itself. Since security proofs are Karp reductions it is natural that the running time of the game is polynomial in λ however, as we shall briefly cover in our discussion of non-falsifiable assumptions this is not always the case.

1.3 Introduction to Proofs/Arguments

What constitutes a proof? What does it mean to prove something? In mathematics/-logic, a proof is a sequence of applications of axioms and previously proven theorems. This notion of a proof might raise a few questions:

What about trivial statements? A mathematical proof convinces us that a statement is true, but it might be non-constructive: the proof might not yield a solution to the problem. Take prime factorization for example: every integer has a prime factorization; by definition. However, it there is no efficient (classical polynomial time) algorithm to recover the prime factorization of an integer. Consequently, the verifier might not be satisfied by the proof that there *exists* a prime factorization of:

```
6250771853489829185205118978883081027626039076873903567516827301
0628257014446164334526839642029665862262223271941617746049101535
4373721896821675401483358502060696184268683410176944812620401112
2833362271530587312324855959685753087422939304414125190688533816
7965234168003511900629645300265679792574058259314829
```

Instead, intuitively, the verifier would like to be convinced, that whomever produced the proof, *knows* the prime factorization. It is not just arbitrary statements like the above – you are unlikely to win any Fields medals for a constructive proof that there exists a prime factorization of the above number. Another classical example is the Gilbert-Varshamov bound [Gil52] in coding theory, which (informally) states *there exists* good (linear) error correcting codes for any alphabet, however, the probabilistic argument of Varshamov [R.57] is non-constructive and a substantial amount of effort is spend in coding theory looking for constructive families of codes trying to meet this *upper bound*. In this case, the verifier would have little interest in a proof that there exists good codes, but would like to be convinced that the prover *knows* a family of good codes.

What about bounded verifiers? In mathematical proofs, we are seldomly explicitly concerned with the computational complexity required to verify a proof, usually, a mathematical proof is in the form of an NP certificate, e.g. a formal proof in a theorem prover [The] [WK19], consisting of a sequence of derivations applying a set of axioms one-by-one reexecuted by the verifier. However, this is quite restrictive: there might be true and interesting propositions whose proof is beyond the computational ability of the verifier. For example, extending the notion of "proof" to allow interaction with the prover and an arbitrarily small soundness error, allows a polynomial time verifier to verify propositions provable in PSPACE [Sha90] [LFKN90] – rather than NP as in the case of traditional proofs. Goldwasser et al. [GKR08] "exponentially scaled down" the result above in term of both prover/verifier complexity, showing that for log-space uniform layered circuits of size S with depth d taking n inputs, a polynomial time prover can prove satisfiability of the circuit to a verifier running in $O(n+d\cdot\mathsf{polylog}(S))$ time; exponentially faster than computing the circuit. The underlying multivariate sum check and its various optimizations [CMT12] [Tha13] [XZZ$^+$19] has subsequently been used extensively in the SNARK literature [Set20] [Tha23].

What about bounded provers? Does a proof need to be unconditionally sound? Namely, can a proof be sound conditioned on cryptographic assumptions – assuming the prover is restricted. Additionally, it might be interesting to study different notions of proofs in idealized models, for instance the PCP model, or the Random Oracle model.

How much do you learn from a proof? Mathematical proofs are often quite insightful, and can give us a deeper understanding of the problem at hand. However there is nothing, a priori, that says that this always needs to be the case: a proof merely needs to convince us that the proposition is true. In fact, it might be interesting to construct proofs which reveal nothing beyond the truth of the proposition. In that case, how should we define this?

Proofs in Cryptography

All of these questions motivate the notion of *proofs* and *arguments* as defined in cryptography. A definition much broader than the classical notion of a proof in mathematics. Broadly speaking, a cryptographic proofs or argument, is a protocol between a prover and a verifier, possibly in some idealized model and possibly under cryptographic assumptions, where the prover tries to convince the verifier of the truth of a proposition. Besides being 'convincing', it can have any number of other properties outlined in the previous section. For instance, it may take the verifier exponentially less time to verify the proof than it took the prover to produce it, it may 'reveal nothing' to the verifier and it may be sound only under cryptographic assumptions – conjectured intractability of some language.

For all of this to make rigorous sense, we need to define these properties formally: so that we can state, in concrete terms, what some protocol achieves and formally proof that it does so. Although constructions are what keeps cryptography alive and

interesting, arguably the most beautiful part of cryptography is its definitions. It is what allows us to confidently construct secure constructions – because we have a clear definition of what it means to be secure. Cryptography is full of elegant definitions, many of which, once understood, can fundamentally change the way we think about the world. I would argue that most brilliant among them is the notions of *simulation* and *extraction*, which capture the opposing notions of *learning nothing* and *knowing something*.

Simulation: What does it mean to learn nothing?

Suppose we wanted to create a protocol in which we would like to argue that a party in the protocol 'learns nothing'. How do we formalize 'learning nothing'?

> "Is is an intresting question how zero-knowledge should be defined.
> ...
> In fact the prover should not reveal *anything* which should help the verifier compute anything much faster than before."
> – Goldwasser, Micali, Rackoff [GMR89]

The way to make this inuition into a security definition is to require the existance of an efficient algorithm, which, given *only blackbox access to the verifier* produces a transcript which is indistinguishable from the real transcript with the honest prover provided with a witness. Inuitively, suppose the verifier could have learned *anything* from communicating with the prover, then he could have learned the same thing by simply simulating the prover himself. Therefore, whether the verifier interacts with the honest prover or not, could not possibly allow him to learn anything he could not have learned by himself. In symbols: there exists a PPT algorithm Sim:

$$\{\mathscr{T} \mid \mathscr{T} \leftarrow \langle \mathsf{V}^*(x), \mathsf{P}(x,w)\rangle\} \approx \{\mathscr{T} \mid \mathscr{T} \leftarrow \mathsf{Sim}^{\mathsf{V}^*}(x)\}$$

Before we continue, let us make a few observations:

1. The verifier is an arbitrary Turing machine. A weaker notion, Honest-Verifier Zero-Knowledge (HVZK), primarily used for public-coin protocols, is that the verifier V^* is the honest verifier V – whose correct behavior might be imperative to ensure that it 'learns nothing'. This definition is motivated by the observation that for public coin protocols, the verifiers behavior can be publicly computed given the public randomness and that restricting the random tape of the verifier, e.g. by sampling it using a Random Oracle (applying Fiat-Shamir) or requiring that the verifier commits to his random tape before the protocol execution, is therefore sufficient to get the stronger notion. For zero-knowledge, a well-known lower bound by Goldreich and Krawczyk [GK90] states that the minimal number of rounds required is 5, however, HVZK can be achieved in 3 rounds.

2. The running time of the simulator is *polynomial in the running time of the (possibly malicious) verifier, not the honest prover*. The reason for this strict

requirement on the simulators efficiency is the intuition for the zero-knowledge definition in the first place: the verifier should not learn anything he could not have computed himself. This intuition goes out the window if the simulator is exponentially more powerful than the verifier, as in the case of the zkSTARKs simulator, in which case he could not possibly produce the transcript himself.

3. By changing the notion of indistinguishablity ($\approx$), we obtain different flavors of zero-knowledge: perfect, statistical and computational zero-knowledge respectively. In the case of computational zero-knowledge, the malicious verifier is restricted to running in time polynomial in the security parameter and the running time of the distinguisher is polynomial in the security parameter.

The reader should note that this definition satisfies the intuition provided above by Goldwasser, Micali and Rackoff [GMR89] in a formal sense: if there exists an algorithm $\mathscr{A}$ which, when interacting with the prover $\mathsf{res} \leftarrow \langle \mathscr{A}(1^\lambda), \mathsf{P}(x,w) \rangle$ can compute results res from some distribution, then there *also exists an algorithm* $\mathscr{A}' = \langle \mathscr{A}(1^\lambda), \mathsf{Sim}(x) \rangle$ which does not interact with the prover and yet produces outputs of an indistinguishable distribution with at most a factor $\mathsf{Poly}(T)$ required to run the simulator (rather than the honest prover): having access to $\mathsf{P}(x,w)$ speeds up the computation of res by at most a polynomial factor. As a corollary, note that for languages in P, the definition is vacuous.

Extraction: What does it mean to Know Something?

Now that we a reasonable idea of what it means for the (possibly malicious) verifier to "learn nothing" during the protocol execution, we should tackle the problem of defining what we mean when we say that the prover "knowns" a solution/witness. The central idea for formalizing this is the notion of extraction, which, informally stated, means that we can convert a prover P^* (some arbitrary Turing machine, not necessarily the honest prover) into another algorithm $\mathscr{E}$, the *knowledge extractor*, which outputs a witness w with (roughly) the same probability and running time as the prover P^* convinces the verifier. Although Goldwasser, Micali and Rackoff [GMR89] also defined a notion of knowledge soundness, the definition widely in use today is due to Bellare and Goldreich [BG93] a few years later. The major difference between the definitions of Goldwasser et. al and Bellare and Goldreich [BG93], is that the latter states that the running time of the extractor may depend inverse polynomially on the *success probability of the malicious prover* (not just its running time), but in return the extractor $\mathscr{E}$ must always produce a witness *for all provers* P^* which succeed with probability $\varepsilon > 2^{-\kappa}$ (the *knowledge error*): even if ε is negligible in the security parameter, meaning the extractor $\mathscr{E}$ might have exponential running time in λ.

Knowledge Soundness. Intuitively, computing a witness for the statement can be achieved in a relativized world with oracle access to a prover P^*, in time that is inverse polynomial in the success probability of the prover. Formally: there exists an extractor $\mathscr{E}$ such that for all provers P^* convincing the verifier with probability $\varepsilon \in (2^\kappa, 1]$. The

extractor $\mathscr{E}^{\mathsf{P}^*} \in \mathsf{Poly}\left(\frac{|x|}{\varepsilon - 2^{\kappa}}\right)$ with oracle (rewinding) access to the prover P^* recovers a witness w with probability 1:

$$\exists \mathscr{E}. \Pr\Big[\langle \mathsf{V}(x), \mathsf{P}^*(1^{\lambda})\rangle = 1\Big] = \varepsilon \implies \mathscr{R}(x,w) = 1 \text{ where } w \leftarrow \mathscr{E}^{\mathsf{P}^*}(x)$$

Where Poly is some fixed polynomial. Several observations should be made about this definition, because it is quite subtle:

1. The probability ε is over the verifiers random tape.
2. The prover P^* is an arbitrary Turing machine, *not the honest prover* P.
3. The order of quantifiers is *very* important: the extractor is chosen *before* the prover, i.e. $\exists \mathscr{E}. \forall \mathsf{P}^* \in \mathscr{P}$.. Which means that the *extractor must work for every prover* and cannot rely on the code of the prover. This is called blackbox/universal extraction, it is the cleanest and most natural definition of knowledge soundness as defined by Bellare and Goldreich [BG93]. However, it is not the only reasonable definition: for instance schemes relying on non-falsifiable so-called 'knowledge assumptions', e.g. Knowledge of Exponent Assumptions [Dam92] will have the quantifiers swapped, requiring the extractor to rely on the code of the prover: meaning the extractor either explicitly takes the prover as input, or, is quantified after the prover ($\forall \mathsf{P}^*. \exists \mathscr{E}.$)
4. The extractor running time is limited by the provers *success probability*, not the provers running time. If on the other hand, the definition allowed the extractor to run in the same time as the prover, then the definition would become vacuous when the prover is unbounded: the extractor could simply brute force a witness. This is one of the subtle nuances in the definition of Bellare and Goldreich [BG93].

Note that the trivial proof, where the prover simply sends the witness w, is clearly knowledge sound with knowledge soundness error $2^{-\kappa} = 0$. Observe also that mathematical proofs in general are not knowledge sound, since convincing the verifier of the truth of a statement does not imply that the prover knows a witness: suppose we have a cyclic group $\mathbb{G} = \langle G \rangle$ where computing discrete logarithms is hard and we wish to prove that a discrete logarithm δ exists for a given element H i.e. $H = [\delta] \cdot G$. Since every element of a cyclic group by definition has a discrete logarithm with respect to the generator G, the mathematical proof of the *existence* of a discrete logarithm is trivial. Therefore the proof above is *sound*, however, it would not allow the extractor $\mathscr{E}$ to compute a witness: it is not *knowledge sound*.

One of the most wondrous connections in cryptography is the counter intuitive fact that the two notions, zero-knowledge and knowledge soundness, though seemingly diametrically opposed, are in fact not mutually exclusive: it is possible to know something, convince someone else that you know it and for them to learn nothing from the interaction. The astute reader might already have noticed that if a protocol is

both zero-knowledge and knowledge sound, then the protocol must necessarily have strictly non-zero knowledge error $2^{-\kappa}$: a malicious prover P^* could guess the verifiers random tape, run the zero-knowledge simulator and hope that the verifier picks the same random tape as in the simulation.

Who Is The Prover Anyway? In cryptography, we talk about extracting from the prover, an arbitrary Turing machine, but in the real-world, where our protocol will be run, the prover could produce responses by *arbitrary means* e.g. asking a friend, spinning the roulette or by some complicated quantum process. Thus the knowledge soundness definition implicitly assumes than any such behavior can be captured as a Turing machine, consequently, philosophically cryptographers subscribe to the Church-Turing thesis [van90] – even asking your friend can be emulated by a (randomized) Turing machine. Therefore the prover P^* is not a person or some computer with an internet connection, but actually a Turing machine emulating the (observable) universe: in an epistemological sense, knowledge soundness then states that it should only be possible to convince the verifier with noticeable probability in *universes* which are also be capable of computing the witness i.e. the witness might *not presently exist* on some hard drive or in someones head, but it *could* be materialized by the universe.

Idealized Models and Non-Falsifiable Assumptions. So far, the story about cryptography outlined in this thesis is an elegant one: that of efficient games, played by standard (non-oracle) Turing machines (adversaries). We call such assumptions *falsifiable* [GK16] [GW11] [Nao03], meaning, at the end of the game, the (efficient) challenger can verify whether the adversary won the game. The assumption states that the adversary has at most negligible winning probability in the game. Unfortunately, not all cryptographic assumptions used in the literature are this elegant. A common example includes Knowledge-of-Exponent Assumption (KEA1) introduced by Damgård [Dam92], which states that for a (family of) cyclic groups $\mathbb{G}$: if there exists a PPT algorithm $\mathscr{A}$ which given $W, G \in \mathbb{G}$, produces $(A,B) \leftarrow \mathscr{A}(W,G)$ with $A = [r] \cdot W$ and $b = [r] \cdot G$, then there *exists* a PPT $\mathscr{A}'$ which given the random tape of $\mathscr{A}$ additionally outputs r: $(A,B,r) \leftarrow \mathscr{A}'(W,G)$. This assumption is non-falsifiable, because the challenger cannot efficiently verify whether the adversary won the game, because doing so requires showing the *non-existence of the efficient algorithm* $\mathscr{A}'$; which cannot be done efficiently. It is easy to see how the assumption that $\mathscr{A}'$ exists is useful when constructing the extractor: without supposing the existence of such an algorithm, the extractor would have to compute the discrete logarithm in $\mathbb{G}$ to recover r, meaning its running time would be exponential in the security parameter. Alternatively, non-falsifiable assumptions might also restrict the set of adversaries considered, e.g. the Algebraic Group Model [FKL18], which restricts the set of adversaries (e.g. malicious provers P^*) to be so-called 'algebraic', meaning the adversary returns a linear combination of previously provided group elements for any group element produced during the protocol. In reality, this implicitly assumes that any efficient adversary relevant to the protocol *could be converted into* an algebraic one – which requires assuming the existence of another PPT algorithm which additionally computes

the linear combination efficiently, analogous to the KEA1 assumption outlined above.

Another option is to have protocols in idealized models, i.e. models in which the adversary/challenger might be oracle machines. Common examples include the Random Oracle Model (ROM) [BR93] where the parties have join query access to an exponentially large table of random bit strings and the Generic Group Model(s) (GGM) of Shoup [Sho97] and Mauer [Mau05] where parties have 'blackbox' access to a group via the oracle, either as a random encoding of a cyclic group (Shoup) or via labels for group elements (Mauer), with Shoup's model being substantial stronger [Zha22] [ZZK22]. In these settings the extractor controls the oracle i.e. may observe queries and produce responses. For protocols proven secure in such idealized models, the natural question is how the security of schemes in the idealized model translates to the real world (standard model) when the oracles are heuristically instantiated with hash functions and groups. Indeed there exists (contrived) protocols in the ROM which are insecure when the random oracle is replaced by *any* hash function [CGH98]. However these feared 'real-world breaks' has not widely manifested themselves in practice, as such, proofs in the ROM/AGM/GGM at least seem like reasonable evidence for the security of the protocol in the standard model when the oracles are heuristically instantiated. Furthermore, in addition to enabling more elegant/natural protocols, the use of idealized models and non-falsifiable assumptions is sometimes unavoidable. Most notably in the case of non-interactive succinct arguments (SNARGs[2]), where the landmark result of Gentry and Wichs [GW11] shows the impossibility of constructing (publicly verifiable adaptively secure) SNARGs from falsifiable assumptions.

How Small Can a Proof/Argument of Knowledge Be? In the case of straight-line extractable proofs, the size of the proof is lower bounded by the size of the witness: the extractor is only provided with a transcript (and possibly a CRS trapdoor), hence the communication between the prover and verifier must be large enough to distinguish each witness for the relation – for obvious information theoretic reasons. However in the cases where the extractor can rely on rewinding, where the protocol is in an idealized model (e.g. the ROM) or depends on non-falsifiable assumptions, the size of the transcript might be much (much!) smaller than the witness. In these cases the extractor can extract a witness either by:

1. Extracting the witness by rewinding.
   The knowledge extractor might reset the prover to a previous state in the protocol execution and can use this to extract the witness piece-by-piece, by resetting the prover to a previous state (without changing the random tape) and observing the resulting transcripts for multiple different challenges. This is the type of extraction outlined in the definition above.

2. Applying knowledge assumptions to the prover.
   Essentially assuming that there exists an efficient algorithm, which in addition to the proof also outputs (values related to) the witness. The non-trivial aspect

[2]Succinct Non-interactive Arguments; the argument is not zero-knowledge and only has regular soundness (no knowledge extractor).

of this strategy lies in the fact that the knowledge assumption is not with regards to the whole proof system[3], but instead wrt. components of the system, e.g. the group operations as in the AGM and then leveraging that to construct a knowledge extractor for the protocol.

3. Observing queries to the oracle in an idealized model.
   The communication/number of queries made by the prover to the oracle might be much greater than the amount of communication between the prover and verifier.

Notation. In the sections that follow it is convenient to have a succinct notation for zero-knowledge proofs which allows specifying which parts remain secret and which are public. Throughout the thesis, we will use the style of Camenisch and Stadler.

Camenisch-Stadler Notation. In the context of zero-knowledge proofs, we will often used Camenisch-Stadler notation [CS97] (knowledge specification sets) to describe relations being proved and which parts constitute the statement and witness. The notation takes the following form $\{(w) : \mathscr{R}(w,x)\}$, with everything not in the tuple being public. As an example, the following is the Camenisch-Stadler notation for a Schnorr proof of knowledge of a discrete logarithm: $\{(\delta) : H = [\delta] \cdot G\}$ – where $x = (H,G) \in \mathbb{G}$ forms the statement and $w = \delta \in \mathbb{Z}_p$ forms the witness. The notation above deviates slightly from the original Camenisch-Stadler notation, we also omit the types of the variables (as above) when clear from the context. In some informal contexts we will also abuse this notation to refer to a function computed inside a multi-party computation protocol (covered below), in which case the witness (tuple) is the private inputs to the function provided by the parties.

1.4 Multiparty Computation and Zero-Knowledge Proofs

Maliciously secure multiparty computation (MPC) is in some sense a generalization of zero-knowledge proofs of knowledge, in which a single party holding an input w to convince a verifier that the function $\mathscr{R}(x,\cdot)$ evaluated to 1 on w, i.e. $\mathscr{R}(x,w) = 1$. MPC expands this by enabling *n parties* $P_1,\ldots,P_N$ holding private inputs $w_1,\ldots,w_n$ to jointly compute a functions $f_1,\ldots,f_n$ on their inputs, and for each party P_i to learn $o_i = f_i(w_1,\ldots,w_n)$., Indeed (interactive) zero-knowledge proofs can be recovered as a special case where $n = 2$, in which player 1 is the prover $P_1 = \mathsf{P}$, player 2 is the verifier $P_2 = \mathsf{V}$, the function f_1 is constant $f_1(w_1,w_2) = 1$ and the function f_2 is the predicate of the relation $f_2(w_1,w_2) := \mathscr{R}(w_1,x)$.

Security Definition of MPC. With this in mind, it should come as no surprise to the reader that the security notions of MPC and ZK are essentially identical; although more complex owning to the generality of MPC. Intuitively the security goal for MPC

[3]You could always make a knowledge assumption about the concrete proof system, meaning that the if a prover computes a proof, then there also exists an efficient algorithm outputting a witness; but that would not be very convincing, we want to prove this from a simpler knowledge assumption.

is that any coalition $\mathscr{I} \in \mathscr{C}$ of corrupted parties from a set of tolerated corruptions $\mathscr{C}$, should be *unable to learn anything beyond what can be inferred from the outputs* of the corrupted parties $\{o_i\}_{i\in\mathscr{I}}$. This is formalized, like zero-knowledge, by the notion of simulation: whereas in the case of zero-knowledge the view of the verifier (the transcript) must be simulatable given only the statement x, in the case of MPC the joint view of the corrupted parties $\{P_i\}_{i\in\mathscr{I}}$ must be simulatable given only the outputs of the corrupted parties $\{o_i\}_{i\in\mathscr{I}}$ for any inputs of the honest parties $\{w_i\}_{i\notin\mathscr{I}}$.

Zero-Knowledge Proofs from MPC. The lines between these notions are further blurred, on one hand, because any MPC protocol can be turned into a 3-round computational zero-knowledge proof with the use of cryptographic commitments [IKOS07], and on the other hand, what is a zero-knowledge proof if not a secure 2PC protocol where only one party has input and the output is a single bit? Conversely, proof systems with multiple non-colluding verifiers, notably the Linear PCP literature [BBC$^+$19], as well as recent zero-knowledge proofs based on vector oblivious linear evaluation correlations [YSWW21], a common preprocessing step in many dishonest majority MPC protocols [KOS16] [HOSS18] burry the lines "in the other direction".

1.5 Disjunctions & Branching Computation

Picking up speed and getting down to the thesis-specific details, the primary topic of this thesis is techniques for efficient privacy preserving computation of branching computation. Deliberately keeping it vague for now, this means that there are two or more possible "options" and the selected option must remain private. In the context of zero-knowledge proofs, this means that the prover knows which "option" was selected from a set, convinces the verifier that the option was selected from the set and that it satisfies some additional constraints, while the verifier learns nothing about the option selected. Broadly speaking, the techniques, namely what constitutes an "option" and what constitutes a "set", can be divided into two categories:

Disjunctions over Statements.

The first category is disjunctions over statements, in which the relation is fixed, the "options" are statements x and the "set" is the set of "allowed" statements $\mathfrak{X}$. In Camenisch-Stadler notation:

$$\{(w,x) : x \in \mathfrak{X} \wedge \mathscr{R}(x,w) = \mathsf{out}\}$$

With "$\mathsf{out} = 1$" for zero-knowledge and the function output in case MPC. Note that the chosen statement x is part of the witness, while the relation (e.g. R1CS matrixes or circuit) is fixed. Examples of such "disjunctions over statements" include set memberships and lookups, further examples are covered in the subsequent section.

Disjunctions over Relations.

The second category is disjunctions over relations, in which the relation is not fixed, the "options" are NP relations $\mathscr{R}$ and the "set" is the set of "allowed" relations/clauses in the disjunction $\mathfrak{R}$. In Camenisch-Stadler notation:

$$\{(w, \mathscr{R}) : \mathscr{R} \in \mathfrak{R} \wedge \mathscr{R}(x, w) = \mathsf{out}\}$$

With "$\mathsf{out} = 1$" for zero-knowledge and the function output in case of MPC. Note that the chosen relation $\mathscr{R}$ is part of the witness, while the instance (i.e. public inputs) is fixed. Examples include 'proofs of partial knowledge' [CDS94] and works such as SubplonK [CGG$^+$23], further examples are covered in the subsequent section.

Relation Between Disjunctions over Relations/Statements.

In the case of both zero-knowledge proofs and multi-party computation, it is possible to generically transform between disjunctions over statements and disjunctions over relations. It is possible to obtain a disjunction over statements from a disjunction over relations, by simply hardcoding the statements into the relations. In the converse direction, it is possible to construct a disjunction over relations using universal circuits, however, this introduces a $O(\log n)$ blowup in the circuit size [Val76] [KS16]. Although not a big problem asymptotically, it is concretely rather costly. Because of this difference in efficiency, with disjunctions over relations generally being more powerful, we find it useful to distinguish between these two (informal) notions when comparing works in the literature.

Landscape of Branching Computation in Zero-Knowledge Proofs.

To provide context for the publications included in this paper, this section includes an overview of the landscape of branching computation in cryptography, in both the context of zero-knowledge proofs (succinct and not), as well as multi-party computation.

Compilers and Sigma Protocols. For Σ-protocols, three move protocols with special soundness, Cramer, Damgard and Schoenmakers [CDS94] introduced an elegant compiler which transforms any Σ-protocol using a maximum distance separable (MDS) code of dimension $\ell - t$ into a Σ-protocol for the t-threshold relation without additional cryptographic assumptions. When instantiated with the parity-check code, this compiler yields a Σ-protocol for disjunctions over relations. The primary 'drawback' of this compiler is that the communication complexity is linear in the number of clauses ℓ. Motivated by the desire to construct ring signatures, Abe et. al [AOS02] proposed a compiler which avoids the need to send both first and last round message of the Σ-protocol as done in CDS, compared to the CDS compiler, the saving is a small concrete factor: roughly a factor of 2. Motivated by techniques from stacked garbling [HK20b] and their possible applications in Σ-protocols, Goel et. al [GGHAK22] recently introduced the "Stacking Sigmas" compiler (included in this thesis) which has

a logarithmic (as oppose to linear) communication complexity in ℓ with small concrete overhead. Unlike CDS, this compiler requires additional cryptographic assumptions and structure of the Σ-protocols being compiled.

Succint Proofs with Branching. Recent work by Goel et. al [GHAKS23] observed that a variant of the Stacking Sigmas compiler [GGHAK22] can be applied to succinct proofs, wherein the complexity of the simulator can often be made poly-logarithmic in the size of the relation. Interestingly, the transformation requires changes to the Fractal Holographic RS-IOPP [COS20]: the verifiers queries to the holographic oracles during the "'holographic lincheck" cannot be simulated for different relations. The resulting proofs are not succinct in the number of clauses ℓ and the prover complexity is linear in ℓ but essentially independent of the size of the unsatisfied clauses. Recently Choudhuri et. al [CGG$^+$23] introduced SubplonK, which is a variation of PlonK [GWC19] which supports disjunctions over "basic-blocks", the concrete improvement over a naive disjunction in PlonK is a relatively small constant factor (≈ 5).

Vector Oblivious Linear Evaluation (VOLE) Based Proofs with Branching. Vector Oblivious Linear Evaluation (VOLE) based proofs [DIO21] [BMRS21] [WYKW21] [YSWW21] [DILO22] use VOLE-correlations as cheap linearly homomorphic commitments: a VOLE-correlation is a pair $(\mathbf{v}, \Delta \cdot \mathbf{v} + \mathbf{m})$ $(\Delta, \mathbf{m})$ – a standard "Information Theoretic MAC" (IT-MAC) often employed in MPC protocols. The primary advantage of VOLE-based commitments is their concrete efficiency when combined with recent advances in OT-extension/Pseudorandom Correlation Generators (PCGs) [IKNP03] [BCGI18] [YWL$^+$20] [Roy22]: committing/opening requires just a few field operations. The primary drawback is that the communication is linear and the protocols are designated verifier (except with the recent work of Baum et. al [BBD$^+$23]). Some works in this literature also address the question of how to efficiently prove disjunctions (with sub-linear communication / computation), notable works include Mac'n'Cheese [BMRS21] which allows the prover to prove that one of ℓ clauses in a (possibly nested) disjunction is satisfied with sub-linear communication, unfortunately, the scheme follows an "execute garbage on all unsatisfied" clauses approach, where after the prover has a zero commitment for the satisfied clause and a random commitment for the unsatisfied clauses, it then proves that at least one of the commitments is zero. The result is that both prover and verifier complexity is linear in ℓ: since both must execute all clauses. Recently the computational complexity of the prover and verifier was improved by Batchman & Robin by Yang et. al [YHH$^+$23] and concurrently in Dora [GHAK23] by Goel et. al. When the same set of clauses are executed many times, the marginal cost of a disjunction over the clauses is $O(\ell + |\texttt{branch}|)$ and $O(|\texttt{branch}|)$ for Batchman and Dora respectively, where $|\texttt{branch}|$ is the maximum size of any clause in the disjunction. Both works are concretely efficient.

Lookup Arguments for SNARKS. Another related line of work is the use of lookup arguments, which enables the prover to do lookups in either static tables (where the verifier/indexer precomputes the table) or dynamic tables (where the prover can insert elements into the tables, subject to some constraints). Notable works include the recent

line of lookup arguments for SNARKs with PlonKish [GWC19] (or CCS [STW23]) arithmetization staring with Arya [BCG+18], Plookup [GW20] and more recent Cached Quotients (qc) [ZBK+22] inspired by the "logarithmic derivative" techniques of Haböck [Hab22] and Eagen [Eag22]. In practical applications, static lookups have primarily been used to speed up non-algebraic operations e.g. foreign field operations, bit decomposition [BCG+18], hash function evaluation and fixed-based elliptic curve operations [HBHW21], while dynamic lookups has found extensive use in virtual machine emulation e.g. efforts to emulate the Ethereum Virtual Machine in a SNARK [zke23], since it enables a prover to simply lookup the result of the current instruction in a table containing only applications of the particular instruction – rather than execute every possible instruction at every step.

Accumulators and Zero-Knowledge Membership Proofs. For large dynamically updated tables/sets, it is desirable that the provers complexity is poly-logarithmic in the set size. The most common approach to this problem is to use a cryptographic accumulator [Bd94] combined with a zero-knowledge proof that the prover knows an opening to an (secret/witness) element in the (public/instance) accumulator. This folklore approach of proving accumulator membership in zero-knowledge has been widely applied, for a variety of accumulators, mostly RSA-based and Merkle trees with various collision resistant hash functions e.g SHA-256 [BCG+14], Pedersen hashes [HBHW21] and 'SNARK-friendly' hash functions (e.g. Poseidon [GKR+21]). Such membership proofs has been used/optimized in cryptocurrency applications e.g. Zerocoin [MGGR13], Zerocash [BCG+14], Veksel [CHA22b] and Curve Trees [CHA22a]. Another notable work is Caulk [ZBK+22], presented as a lookup argument, but included here under membership proofs, because the provers complexity is sublinear in the table size. However, one notable drawback is that like the other lookup arguments, the structured reference string is linear in the table size – as oppose to polylogarithmic as for the accumulators outlined above.

SNARKs for RAM programs. SNARKs for C [BCG+13] which proves execution of a very simple RAM machine (TinyRAM) inside a SNARK. An alternative approach was taken in Buffet [WSR+15] which branches over basic blocks directly (a basic block is a sequence of instructions with no branches), this avoids the need to repeated prove the fetch-and-decode part of the TinyRAM machine. The disadvantage of this approach is that the setup is not universal, i.e. the CRS depends on the program being executed as oppose to the approach taken by Ben-Sasson et. al. [BCG+13], in which the CRS only depends on an upper bound on the execution time and the architecture of the machine (TinyRAM). Recently, industry efforts motivated by blockchain applications have constructed SNARKs for far more complicated instruction sets, including the RISC-V instruction set [JB23] and the Ethereum Virtual Machine [zke23]. The practicality of these efforts is largely due to the use of lookup arguments, "SNARK-friendly" hash functions and the substantial improvements in the SNARK literature during the last ten years in general.

Non-Uniform IVC and PCD. In 2008 Valiant [Val08] introduced Incrementally Verifiable Computation (IVC) which enables a prover to incrementally prove the

execution of a Turing machine: given a proof of the execution of the first n steps, the prover can efficiently extend the proof to $n+1$ steps. The construction relies on a tree of Micali CS proofs [Mic00], which in turn are constructed from extractable PCPs and random oracles. From a theoretical perspective, this leads to a problem: as CS proofs are used to prove the execution of the CS proof verifier, however the CS proof verifier is not a standard Turing machine, but rather an oracle Turing machine with access to a random oracle. Heuristically, this is overcome by instantiating the random oracle with a cryptographic hash function and assuming that the resulting CS proof is a SNARK in the standard model. Chiesa and Tromer generalized the notion of IVC to the setting of arbitrary graphs of computation with the introduction of Proof-Carrying Data (PCD) [CT10], however, to avoid the non-blackbox use of a random oracle in Valiant's construction and to achieve a very efficient online extraction required to ensure that the running time of the extractor does not grow exponentially in the recursion depth (which was logarithmic in Valiant's construction), their PCD construction relies on a signed-input-and-randomness oracle which can be seen as a form of random oracle which additionally signs the query/response pairs enabling a verifier to check the response without access to the oracle. Subsequently Bitansky et. al [BCCT13] showed how to bootstrap any preprocessing SNARK into a PCD scheme, however the resulting scheme only works for constant depth recursion as the extractor for the PCD scheme is obtained via recursively composing the SNARK extractor, leading to an exponential growth in the running time of the knowledge extractor. The first "practical" (read "runs on a computer") application of this framework was obtained by Ben-Sasson et. al [BCTV14a] from cycles of pairing friendly curves, the primary hurdle in terms of efficiency is that the only such cycles of curves have a low embedding degree k, meaning the field of the curve $\mathbb{F}_q$ must be large to obtain a reasonable security level in the presence of an efficient mapping into $\mathbb{F}^*_{q^k}$, in practice around > 700 bits for MNT curves at ≈ 100 bits of security, which makes operations on the curve expensive, in particular the pairing operation which is proven in-SNARK. In 2019, Ben-Sasson et. al [BBHR19] applied the same techniques to STARKs [BBHR18], similarly Chiesa et. al [COS20] constructed a holographic RS-IOP also compiled with the FRI proximity testing, both construct a PCD scheme without the need for a trusted setup with practical efficiency by heuristically instantiating the random oracle with "SNARK/STARK-friendly" hash functions [AD18] [GKR$^+$21]. Recently, the introduction of split/atomic accumulation based PCD/IVC protocols starting with Halo [BGH19] and the generalization of accumulation schemes[4] [BCMS20] [BCL$^+$21]. has lead to a cambrian explosion of practical IVC/PCD schemes from (pairing free) cycles of elliptic curves, notable works include Nova [KST22] (IVC) & SuperNova (Non-Uniform IVC) [KS22], HyperNova [KS23] (PCD for CCS relations) and Protostar [BC23]. From a theoretical perspective, one interesting aspect of these recent protocols protocols is that (unlike the earliest works) they all suffer from an exponential loss in knowledge soundness in the recursion depth; since the extraction technique relies on recursive rewinding.

[4] No relation to cryptographic accumulators.

Nonetheless, the protocols does not seem to suffer any loss of security in practice and has been deployed in blockchain-related applications [min23].

Landscape of Branching Computation in MPC.

For MPC branching computation is often much efficient than in Zero-Knowledge Proofs, since no single party knows which clause is taken and on which input it is executed.

Private Function Evaluation. In Private Function Evaluation (PFE) [KM11, MS13, MSS14, HKRS20] a single party holds a description of a function f, and the other parties hold inputs $x_1, \ldots, x_n$. The goal of PFE is to compute $y = f(x_1, \ldots, x_n)$ without revealing f or any of the x_i – beyond what can be simulated given y. One can think of a setting where one of the parties holds some proprietary function f e.g. a machine learning model, and wishes to make this function available to the other parties without revealing the function. Private function evaluation differs from the settings we consider in this thesis in that the function is *unconstrained* and *known* to a single party, therefore, PFE is in may ways easier than the setting of branching over relations. Nevertheless, the work of Goel et. al [GHAHJ22] included in this thesis uses techniques inspired by the PFE construction of Mohassel and Sadeghian [MS13].

Communication Efficient Branching MPC. In the context of Garbled circuits, Heath and Kolesnikov [HK20b] introduced the idea "stacked garbling". Which derives from the observation that the garbled circuit, from the evaluators perspective, is just a (pseudo) random string, assuming the circuit is encoded by simply concatenating the garbled tables ("topology-decoupling circuit garbling" [Kol18]), thus its distribution reveals nothing about the circuit. Their technique results in a computational overhead of $O(\ell^2)$: as every active clause results in $\ell - 1$ garbled garbage outputs, which the prover must compute to encode the gadget. Subsequently this substantial computational overhead was reduced to $O(\ell \log \ell)$ [HK21] by the same authors using memorization techniques. Work by Heath et. al (MOTIF) [HKP20] improves the number of public key operations in GMW [GMW87] when computing multiple branches in parallel. The asymptotic communication complexity of MOTIF is $O(n^2 \cdot |C| \cdot \ell)$, where $|C|$ is the size of the circuit, ℓ is the number of clauses and n is the number of parties. Hence the saving over the naive approach is only concrete. At Eurocrypt 2022, Goel et. al [GHAHJ22] constructed a concretely efficient generic protocol which has $O(f(n) \cdot (|C| + \ell))$ communication complexity, where $f(n)$ is the communication complexity of executing a multiplication gate, in particular, for the case of GMW, $f(n) = O(n^2)$ and for CDN [CDN01] $f(n) = O(n)$. The techniques are derived from a private function evaluation protocol, in which the function holder has been distributed. This paper is included in the thesis.

MPC for RAM / Oblivious Data Structures An obvious way to attain "branching" in MPC is by constructing MPC for a branching model of computation, e.g. Turing machines or RAM programs, rather than circuits. Towards this end, oblivious data structures are a natural building block, namely oblivious stacks in the case of Turing

machines and oblivious RAM in the case of RAM programs. Oblivious RAM [GO96a] (ORAM), introduced by Goldreich and Ostrovsky in 1996, initially with the motivation of hiding the access pattern of a RAM program running inside an enclave. Concretely efficient instantiations [SvS+13] of the primitive has been deployed by industry in this role [Con22]. More broadly ORAM is a two party protocol between a 'client' (the enclave in the example above) and a 'server' (the untrusted memory), in which the server maintains a large encrypted RAM while the client, with only polylogarithmic storage, can access the RAM obliviously: without revealing the access pattern of the RAM. In order to leverage ORAM in the context of MPC, the client must be run inside the MPC, motivated by this, there are two natural avenues:

1. Design an ORAM with a client that can be efficiently computed inside MPC [WCS15].
2. Construct distributed variants of ORAM directly: Distributed Oblivious RAM.

At Eurocrypt 2013, Lu and Ostrovsky [LO13] introduced Garbled RAM (GRAM), which evaluates a ORAM client inside a garbled circuit, however this requires non-blackbox use of a PRF – evaluated inside a garbled circuit, which makes the construction impractical. Lately EpiGram [HKO22] and NanoGram [PLS23] removed the need to do cryptography inside the garbled circuit and obtained the first concretely efficient garbled RAM. Note that due to ORAM lower bounds [Gol87, GO96b, LN18] any approach based on ORAM necessarily incurs $O(N\log(N))$ complexity. In a Distributed ORAM (DORAM), two or more parties share the state of the RAM and *neither party* learns the access pattern or contents – which is secret shared among the parties. Notable recent works includes a sequence of works optimizing 3PC DORAMs, e.g. Ramen [BPRS23], GigaDORAM [FOSZ23].

MPC for Branching Programs. Some works have also considered MPC for the (much) more restricted model of (deterministic) branching programs. This line of work includes the theoretical work of Ishai and Paskin [IP07] which demonstrates how to homomorphically evaluate branching programs on encrypted inputs. The most notable the work in this area is that of Boyle, Gilboa and Ishai [BGI16] which constructed Function Secret Sharing (FSS) for branching programs from DDH. All these techniques have very large concrete overheads.

Chapter 2

Private Branching Computation

In this section we provide a brief overview of the works included in the thesis and the most central techniques employed.

2.1 Disjunction Compilers: Stacking Sigma Protocols

The first paper included in the thesis is *Stacking Sigmas: A Framework to Compose* Σ*-Protocols for Disjunctions* [GGHAK22].

Preqrequisites. The paper arose from the observation that in many Σ-protocols the distribution of the final message z is independent of the statement x. Furthermore, given a third round message z and a challenge c, the simulator often works by *deterministically computing* the unique accepting first round message a from (z, c, x). In fact this optimization is readily used throughout the Σ-protocol literature, e.g. in lattice based signatures [LDK+22] where sending the challenge c and last round message z, then running the simulator and checking against c, is much smaller than sending (a,z) and computing $c = \mathsf{H}(x,a)$.

Key Observation. If a Σ-protocol has these properties, then it is possible to produce simulated transcripts $(a_1,c,z),\ldots,(a_\ell,c,z)$ sharing the same challenge c and final message z, but for distinct statements $x_1,\ldots,x_n$. This is the key observation of the paper. The stacking compiler, like that of Cramer et al. [CDS94], works by allowing the prover to simulate $\ell-1$ of the transcripts, however a key difference is that the z used to simulate unsatisfied clauses is the one produced by the honest prover on the satisfied clause, furthermore, the challenge c is the same for all transcripts. This creates a chicken-and-egg problem: the prover must send $a_1,\ldots,a_\ell$ to the verifier in the first round before learning c (for soundness), however, he cannot simulate the first-round message a for any unsatisfied clause without first knowing the challenge c.

Partially-Binding Vector Commitments. This issue is solved using partially-binding vector commitments: a hiding commitment to vectors $(a_1,\ldots,a_\ell)$ which is equivocal in *all but one coordinate*. With this type of commitment the prover can now commit to the first-round messages a_α for the satisfied clause x_α by sending

$a = \mathsf{Com}(\bot, \ldots, a_\alpha, \ldots, \bot; r)$ to the verifier which responds with a challenge c. After obtaining the challenge c the prover computes the third round message z for the satisfied clause x_α and simulates the remaining $\ell - 1$ transcripts obtaining the full list of first-round messages $a_1, \ldots, a_\ell$. Finally the prover equivocates the commitment a to the $(a_1, \ldots, a_\ell)$ obtaining commitment randomness r'. The prover sends (z, r') to the verifier. The verifier simulates all ℓ transcripts, then recomputes the commitment $a = \mathsf{Com}(a_1, \ldots, a_\ell; r')$ and checks that it matches the commitment received from the prover. The resulting protocol is an argument, since the 1-of-ℓ binding of the commitment is computational.

2.2 Set Memberships: Curve Trees

The second work in this thesis *Curve Trees: Practical and Transparent Zero-Knowledge Accumulators*. Proposed concrete improvements to proving accumulator membership inside commit-and-prove zero-knowledge.

The Curve-Tree Accumulator. The paper exploits cycles of elliptic curves: pairs of (prime order) elliptic curves $\mathbb{E}_{(\text{evn})}[\mathbb{F}_q]$ and $\mathbb{E}_{(\text{odd})}[\mathbb{F}_p]$ such that $|\mathbb{E}_{(\text{evn})}[\mathbb{F}_q]| = p$ and $|\mathbb{E}_{(\text{odd})}[\mathbb{F}_q]| = q$, i.e the number of points on one curve is the modulo defining the field on which the points of the other resides. This structure is used to build an "algebraic" Merkle tree [Mer88] using Pedersen commitments alternating over the curves as compression functions:

- The hash of $\mathbf{v} \in \mathbb{F}_p^d$ for even levels of the tree is $h = \langle \mathbf{v}, \mathbf{G}_{(\text{evn})} \rangle \in \mathbb{E}_{(\text{evn})}[\mathbb{F}_q] \subseteq \mathbb{F}_q^2$.
- The hash of $\mathbf{v} \in \mathbb{F}_q^d$ for odd levels of the tree is $h = \langle \mathbf{v}, \mathbf{G}_{(\text{odd})} \rangle \in \mathbb{E}_{(\text{odd})}[\mathbb{F}_p] \subseteq \mathbb{F}_p^2$.

Note that this avoids any "bit-decompositions" or "non-algebraic" operations, since the range of one hash function $\mathbb{F}_q^2$ is the included in the domain of the other $\mathbb{F}_q^d$ and vice-versa. Additionally, the commitments/hashes in the tree can easily be made perfectly hiding by adding a random blinding factor to each level.

Traversing The Tree with Commit-and-Prove. This structure allows proving membership in the accumulator efficiently using two commit-and-prove SNARKs for Pedersen commitments: a SNARK over $\mathbb{F}_p$ capable of opening Pedersen commitments in $\mathbb{E}_{(\text{evn})}[\mathbb{F}_q]$ used to handle even levels of the tree and another SNARK over $\mathbb{F}_q$ capable of opening Pedersen commitments in $\mathbb{E}_{(\text{odd})}[\mathbb{F}_p]$ used to handle odd levels of the tree. With these, we can prove membership in the accumulator by traversing the tree from root towards the desired leaf:

The prover opens the root $C \in \mathbb{E}_{(\text{evn})}[\mathbb{F}_q]$ of the tree using the CP-SNARK over $\mathbb{F}_p$ inside the SNARK, the preimage of this commitment is a set of points/Pedersen commitments on $\mathbb{E}_{(\text{odd})}[\mathbb{F}_p]$: one for each child of the root. The prover selects the desired child (leading to the leaf) and exposes this as public input to the SNARK.

The approach outlined above is however *not zero-knowledge*, since the prover reveals the child commitment, thus leaking the path to the leaf. In order to remedy this,

the prover rerandomizes the child commitment inside the SNARK before exposing it as public input to the SNARK: this requires proving a scalar multiplication over $\mathbb{E}_{(\text{odd})}[\mathbb{F}_p]$ inside the $\mathbb{F}_p$-SNARK.

Efficiency. Compared to Merkle trees with SNARK-friendly hashes, e.g. Poseidon, the number of constraints in Curve Tress is only marginally ($\approx 14\%$) smaller and the primary advantage is more conservative cryptanalysis: Curve trees relies on discrete log on elliptic curves and does not need to introduce a hardness assumption on a concrete hash function instantiated over the particular field that the SNARK operates over. For benchmarking, we instantiate the commit-and-prove using Bulletproofs, however, the scheme can be instantiated using any CP-SNARK for Pedersen commitments e.g. a commit-and-prove variant of PlonK [GWC19] / Marlin [CHM+20] using a folding argument, or, by using a pairing friendly cycle.

2.3 Oblivious Branching Computation: Branching MPC

The final work in this thesis is *Secure Multiparty Computation with Free Branching* [GHAHJ22], which proposes a new approach to oblivious branching computation in MPC protocols, allowing conditional execution of 1-of-ℓ sub-circuits inside MPC without revealing the branch taken. More specifically, the protocol allows the parties to oblivious branch over a value and execute one of ℓ functions on the same (secret) input, this type of "switch statement" is illustrated in Figure 2.1 using pseduocode.

```
switch (option) {
    case 0:
        out = function0(input);
    case 1:
        out = function1(input);
    case 2:
        out = function2(input);
    ...
}
```

Figure 2.1: The type of conditional statements supported by Goel et al. [GHAHJ22].

Hidign Gate Types. Assume each clause (function) is implemented as an arithmetic circuit (addition and multiplication gates) with fan-in 2 and arbitrary fan-out. The first observation is that *hiding the gate type is easy*, at the cost of possibly losing "free" addition: simply use universal gates, e.g. a gate of the form:

$$\mathsf{res} = \left(y_{\mathsf{left}} + y_{\mathsf{right}}\right) + \mathsf{type} \cdot \left(y_{\mathsf{left}} \cdot y_{\mathsf{right}} - \left(y_{\mathsf{left}} + y_{\mathsf{right}}\right)\right)$$

When $\mathsf{type} = 0$, the gate is an addition gate ($\mathsf{res} = y_{\mathsf{left}} + y_{\mathsf{right}}$), and when $\mathsf{type} = 1$ the gate is a multiplication gate ($\mathsf{res} = y_{\mathsf{left}} \cdot y_{\mathsf{right}}$). The expression can be evaluated using two multiplications and three additions. Assuming the parties hold a unary representation of active clause, i.e. a "one-hot" encoding of the active clause: $(b_1, \ldots, b_\ell)$ such that $b_i = 1$ if and only if the i-th clause is active and otherwise $b_i = 0$, then the parties can compute the (hidden) type of each gate in the active clause using linear operations:

$$\mathsf{type} = \sum_{i=1}^{\ell} b_i \cdot \mathsf{type}_i$$

Where type_i is the type of the gate in the i-th clause: a public constant.

Hiding Circuit Topology. The far more challenging problem is to hide the wiring/-topology of the active clause. The paper proposes different approaches, but for this overview we focus on the multi-round (optionally maliciously secure) protocol which draws inspiration from private function evaluation (PFE) protocol of Mohassel and Sadeghian [MS13]: in their protocol, each wire is assigned two random masks, input masks: $\mathsf{in}_1, \ldots, \mathsf{in}_W$ and output masks: $\mathsf{out}_1, \ldots, \mathsf{out}_W$. In the PFE protocol, with the help of the function holder, who knows the function topology $\pi : [W] \to [W]$, mapping from input wire indexes to (previously defined) output wire indexes[1] the parties compute differences $\Delta_w = \mathsf{in}_w - \mathsf{out}_{\pi(w)}$ inside MPC and reveal Δ_w to the function holder. The circuit is then evaluated gate-by-gate:

- **Output Masking.** After evaluating a gate, the parties use the output mask of the output wire to mask the output value: computing $u_w = \mathsf{res} + \mathsf{out}_w$ and reconstructs u_w to everyone.
- **Input Unmasking.** Before evaluating a gate, the output masking of the input wires must be removed: the function holding party computes $A = u_{\pi(a)} + \Delta_a$ and $B = u_{\pi(b)} + \Delta_b$, where $u_{\pi(a)}$ and $u_{\pi(b)}$ are the masked outputs. The function holding party sends A and B to all parties, which can then compute secret sharings of the unmasked inputs locally.

This challenge solved by the protocol of Goel et al. [GHAHJ22] is essentially executing the function holder of the PFE protocol inside MPC without the having communication grow with the set of clauses and without introducing an impractical amount of overhead. The central challenge is that of oblivious computing the "shuffled differences", this is solved by computing:

$$\Delta_w = \mathsf{in}_w - \sum_{i=1}^{\ell} b_i \cdot \mathsf{out}_{\pi_i(w)}$$

Inside MPC. However, doing so naively would require ℓ multiplications per wire: meaning the communication would grow linearly with ℓ, defeating the purpose of the

[1] An "this input comes from this output" mapping: wire w, should be assigned the value from the wire $\pi(w)$.

protocol. To avoid this, observe that if we look at the "vector" Δ of all the Δ_w's, we can write:

$$\Delta = \mathsf{in} - \sum_{i=1}^{\ell} b_i \cdot \mathsf{V}^{(i)}$$

With $\forall w.\mathsf{V}_w^{(i)} = \mathsf{out}_{\pi_i(w)}$ which the parties can compute locally by simply permuting their shares of out according to the public permutation π_i – in the PFE protocol the permutation is secret, but here it is public as the set of possible clauses is known. With this formulation the multiplications take the form of a vector-scalar product, to optimize the computation, the parties compute a linearly homomorphic encryption each b_i. With this they can compute an secret sharing of the encryption $[\Delta]$ of Δ using linear operations on their shares:

$$[\Delta] = \mathsf{in} - \sum_{i=1}^{\ell} [b_i] \cdot \mathsf{V}^{(i)}$$

The ciphertext $[\Delta]$ is then converted back into a secret sharing to execute the online phase of the protocol. This trick can be seen as a "vectorized version" of the central technique used by Cramer, Damgård, and Nielsen [CDN01]. During benchmarking the homomorphic encryption is instantiated using a variant of BFV [FV12, Bra12] though significantly simplified by the fact that the parties only need to compute a single linear operation. For the field $\mathbb{F}_{2^{16}+1}$, the homomorphic operations account for a negligible fraction of the total runtime compared to the gate-by-gate online phase of the protocol.

Chapter 3

Works Not Included in the Thesis

In addition to the three papers included in the thesis, the candidate has also contributed to the following published manuscripts during his PhD studies at Aarhus University:

On Valiant's Conjecture: Impossibility of Incrementally Verifiable Computation from Random Oracles

Authors:
Mathias Hall-Andersen (Aarhus University),
Jesper Buus Nielsen (Aarhus University)

Published:
Eurocrypt 2023.

Description:
Valiant's construction of incrementally verifiable computation (IVC) relies on non-blackbox use of the random oracle. In his paper Valiant conjectured that "standard applications of random oracles do not appear to help". This paper gives strong evidence for this conjecture, in particular, proving the conjecture when the IVC scheme is zero-knowledge (simulatable in the random oracle model).

Speed-Stacking: Fast Sublinear Zero-Knowledge Proofs for Disjunctions.

Authors:
Aarushi Goel (NTT Research),
Nicholas Spooner (University of Warwick),
Mathias Hall-Andersen (Aarhus University),
Gabriel Kaptchuk (Boston University)

Published:
Eurocrypt 2023.

Description:
The paper builds upon the "stacking sigmas" framework, by observing that for succinct proofs the simulator can be exponentially faster than the

honest prover. This is exploited to prove disjunctions over clauses, where the provers running time is that of the largest clause (rather than the sum of all clauses).

Veksel: Simple, Efficient, Anonymous Payments with Large Anonymity Sets from Well-Studied Assumptions.

Authors:
Matteo Campanelli (Protocol Labs),
Mathias Hall-Andersen (Aarhus University)

Published:
AsiaCCS 2023.

Description:

Efficient Set Membership Proofs using MPC-in-the-Head.

Authors:
Aarushi Goel (NTT Research),
Matthew Green (Johns Hopkins University),
Mathias Hall-Andersen (Aarhus University)
Gabriel Kaptchuk (Boston University)

Published:
PoPETs/PETS 2023.

Description:
The paper constructs a simple set membership proof from MPC-in-the-head style zero-knowledge proofs and vector commitments using cut-and-choose. The membership is plausibly post-quantum: relying only on the (classical) random oracle model and the security of the MPCitH protocol (which can be constructed from pseudo random functions).

Efficient Proofs of Software Exploitability for Real-World Processors.

Authors:
Matthew Green (Johns Hopkins University),
Mathias Hall-Andersen (Aarhus University),
Eric Hennenfent (Trail of Bits),
Gabriel Kaptchuk (Boston University),
Benjamin Perez (Trail of Bits),
Gijs Van Laer (Johns Hopkins University).

Published:
PoPETs/PETS 2023.

Description:

Describes techniques and software for proving the existence of software vulnerabilities on the `MSP430` micro controller using MPC-in-the-head. Developed as part of the DARPA SIEVE program.

Count Me In! Extendability for Threshold Ring Signatures.

Authors:

Diego F. Aranha (Aarhus University),
Mathias Hall-Andersen (Aarhus University),
Anca Nitulescu (Protocol Labs),
Elena Pagnin (Chalmers University),
Sophia Yakoubov (Aarhus University)

Published:

PKC 2022.

Description:

The paper proposed extendability for threshold ring signatures: given a threshold ring signature a new signing party can non-interactively extend the ring and/or increase the threshold if his public key is in the ring. A motivating application is "count me in" where a party wants to later anonymously endorse a statement already signed by a set of parties.

Game Theory on the Blockchain: A Model for Games with Smart Contracts.

Authors:

Mathias Hall-Andersen (Aarhus University),
Nikolaj Schwartzbach (Aarhus University)

Published:

International Symposium on Algorithmic Game Theory (SAGT 2021).

Description:

Inspired by the game theory which arises from settings wherein actors/-players can deploy arbitrary programs which may inspect each other, the paper formalizes a generalization of reverse Stackleberg games (capturing both Stackleberg and reverse Stackleberg games as special cases). The paper proves a number of positive result (efficient algorithms for computing equilibria) and negative results (hardness results) for other classes of games.

Part II

Included Publications

Chapter 4

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

Aarushi Goel, Gabriel Kaptchuk, Mathias Hall-Andersen, Matthew Green.

Orignally published at Eurocrypt 2022.

4.1 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, BCTV14b, Gro16, KKW18, BBB$^+$18, BCR$^+$19, HK20b], resulting in concretely efficient zero-knowledge protocols that are now being used in practice [BCG$^+$14, Zav20, se19].

Zero-knowledge for Disjunctive Statements. There is a long history of developing zero-knowledge techniques for *disjunctive statements* [CDS94, AOS02, GMY03]. Disjunctive statements comprise of several *clauses* that are composed together with a logical "OR." These statements also include conditional clauses, *i.e.* clauses that would only be relevant if some condition on the statement is met. The witness for such statements consists of a witness for one of the clauses (also called the *active* clause), along with the index identifying the active clause. Disjunctive statements occur commonly in practice, making them an important target for proof optimizations. For example, disjunctive proofs are often also used to give the prover some degree of privacy, as a verifier cannot determine which clause is being satisfied. Use cases

include membership proofs (*e.g.* ring signatures [RST01]), proving the existence of bugs in a large codebase (as explored in [HK20b]), and proving the correct execution of a processor, which is typically composed of many possible instructions, only one of which is executed at a time [BCG$^+$13].

An exciting line of recent work has emerged that reduces the communication complexity for proving disjunctive statements to the size of the largest clause in the disjunction [Kol18, HK20b]. While succinct proof techniques exist [Gro10, GGPR13, BCTV14b, Gro16], known constructions are plagued by very slow proving times and often require strong assumptions, sometimes including trusted setup. These recent works accept larger proofs in order to get significantly faster proving times and more reasonable assumptions — while still reducing the size of proofs significantly. Intuitively, the authors leverage the observation that a prover only needs to honestly execute the parts of a disjunctive statement that pertain to their witness. Using this observation, these protocols modify existing proof techniques, embedding communication-efficient ways to "cheat" for the inactive clauses of the disjunctive statement. We refer to these techniques as *stacking* techniques, borrowing the term from the work of Heath and Kolesnikov [HK20b].

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

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

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

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

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

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

Our Contributions.

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

Self-Stacking Compiler. First, we present our basic compiler, which we call a "self-stacking" compiler. This compiler composes several instances of the *same* Σ-protocol, corresponding to a particular language into a disjunctive proof. The resulting protocol

[1] Although concrete efficiency is a central element of our work, applying our compiler to applications is not our focus. The details of this ring signature construction can be found in Section 4.10.

has communication complexity proportional to the communication complexity of a single instance of the underlying protocol. Specifically, we prove the following theorem:

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

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

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

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

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

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

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

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

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

is very different and does not consider communication saving, but our work makes use of similar (k,n)-equivocal commitments (called 'partially-binding commitments' here) to obtain communication savings rather than delayed instance specification.

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

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

4.3 Technical Overview

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

Recall that Σ-protocols are three-round, public-coin zero-knowledge protocols, where the prover sends the first message. In the second round, the verifier sends a random "challenge" message to the prover, that only depends on the random coins of the the verifier. Finally, in the third round, the prover responds with a message based

[2]We assume that basic, practice-oriented optimizations have already been applied to the Σ-protocols in question. For instance, we assume that only the minimum amount of information is sent during the third round of protocol. Hereafter, we will ignore these trivial modifications and simply say "without requiring modification." Note that these modifications truly are trivial: the parties only need to repeat existing parts of the transcript in other rounds. We discuss this in the context of MPC-in-the-head protocols in Section 4.6.

on this challenge. Based on this transcript the verifier then decides whether to accept or reject the proof.

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

Our Approach. In this work we extend the *cheat and re-use* approach to design a framework for compiling Σ-protocols into a communication-efficient Σ-protocol for disjunctive statements without requiring modification of the underlying protocols. Specifically, we are interested in reducing the number of *third round messages* that a prover must send to the verifier, since the third round message is typically the longest message in the protocol. Intuition extracted from prior work leads us to a natural high-level template for achieving this goal: *Run individual instances of* Σ*-protocols (one-for each clause in the disjunction) in parallel, such that only one of these instances (the one corresponding to the active clause) is honestly executed, and the remaining instances re-use parts of this honest instance.*

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

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

Our first idea is to defer the selection of first round messages for the inactive clauses until after the verifier sends the challenge (*i.e* in the third round of the compiled protocol), while requiring that the prover select a first round message honestly for the active clause (*i.e* in the first round of the compiled protocol). To do this, we introduce

a new notion called *non-interactive, partially-binding vector commitments*.[3] These commitments allow the committer to commit to a vector of values and equivocate on a hidden subset of the entries in the vector later on. For instance, a 1-out-of-ℓ binding commitment allows the committer to commit a vector of ℓ values such that that one of the vector positions (chosen when the commitment is computed) is binding, while allowing the committer to modify/equivocate the remaining positions at the time of opening. For a disjunction with ℓ clauses, we can now use this primitive to ensure that the prover computes an honest transcript for at least one of the Σ-protocol instances as follows:

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

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

Challenge 2: How will the prover re-use the active transcript? The above approach overcomes the first challenge, but doesn't achieve our goal of reducing the communication complexity of the compiled Σ-protocol. Next, we need to find a way to somehow re-use the honest transcript of the active clause. Our key insight is that for many natural Σ-protocols, it is possible to simulate *with respect to a specific third round message*. That is, it is often easy to simulate an accepting transcript for a given challenge and third round message. This allows the prover to create a transcript for the inactive clauses that share the third round message of the active clause. In order for this compilation approach to work, Σ-protocols must satisfy the following properties (stated here informally):

– *Simulation With Respect To A Specific Third Round Message:* To re-use the active transcript, the prover simulates *with respect to the third round message of the active*

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

[4] We also explore a construction that is half the size and leverages random oracles in Section 4.16.

transcript. This allows the prover to send a single third round message that can be re-used across all the clauses. More formally, we require that the Σ-protocol have a simulator that can reverse-compute an appropriate first round message to complete the accepting transcript for any given third round message and challenge. While not possible for all Σ-protocols, simulating in this way—i.e., by first selecting a third round message and then "reverse engineering" the appropriate first round message—is actually a common simulation strategy, and therefore possible with most natural Σ-protocols. In order to get communication complexity that only has a logarithmic dependence on the number of clauses, we additionally require this simulator to be deterministic.[5] We formalize this property in Section 4.6.

– *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 4.6.

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

Summary of our Stacking Compiler. Having outlined our main techniques, we now present a detailed description of our compiler for 2 clauses, as depicted in Figure 4.1 (similar ideas extend for more than 2 clauses). The right (unshaded) box represents the active clause and the left (shaded) box represents the inactive clause. Each of the following numbered steps refer to a correspondingly numbered arrow in the figure: (1) The prover runs the first round message algorithm of the active clause to produce a first round message a_2. (2) The prover uses the 1-of-2 binding commitment scheme to commit to the vector $\mathbf{v} = (0, a_2)$. (3) The resulting commitment constitutes

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

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

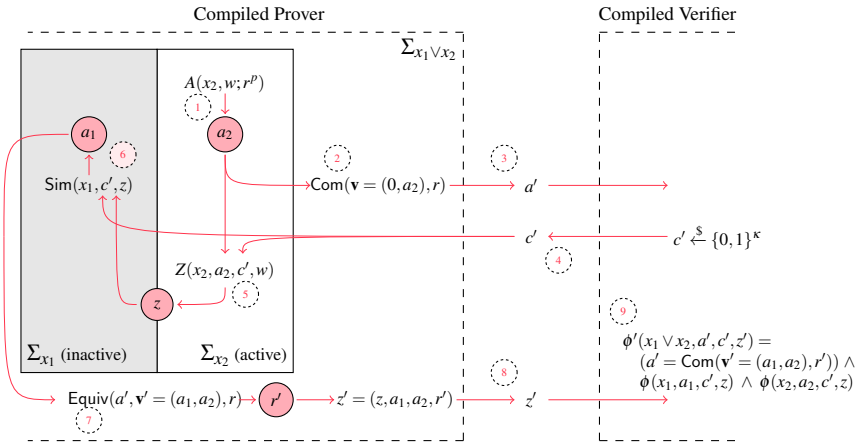

Figure 4.1: High level overview of our compiler applied to a Σ-protocol $\Sigma = (A, C, Z, \phi)$ over statements x_1 and x_2. Several details have been omitted or changed to illustrate the core ideas more simply. The red circle contains a value used in the first round, while purple circles contain values used in the third round. We include a_1 and a_2 in the third round message for clarity; in the real protocol, the verifier will be able to deterministically recompute these values on their own.

the compiled first round message a'. (4) The challenge c' is created by the verifier. (5) The prover generates the third round message z for the active clause using the first round message a_2, the challenge c', and the witness w. (6) The prover then uses the simulator for the inactive clause on the challenge c' and the honestly generated third round message z to generate a valid first round message for the inactive clause a_1. (7) The prover equivocates on the contents of the commitment a' – replacing 0 with the simulated first round message a_1. The result is randomness r' that can be used to open commitment a' to the vector $\mathbf{v}' = (a_1, a_2)$. (8) The compiled third round message consists of honestly generated third round message z, the randomness r' of the equivocated commitment, and the two first round messages a_1, a_2.[7] (9) The verifier then verifies the proof by ensuring that each transcript is accepting and that the first round messages constitute a valid opening to the commitment a'.

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

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

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

communication complexity of the underlying stackable Σ-protocol, when executed for the largest clause. In the technical sections, we show that the resulting protocol is itself "stackable", it can be recursively compiled. This reduces the communication complexity to $O(\mathtt{CC}(\Sigma) + \log(\ell) \cdot \lambda)$.

Stackable Σ-Protocols. While not all Σ-protocols are able to satisfy the first two properties that we require, we show that many natural Σ-protocols like Schnorr [Sch90], and Guillio-Quisquater [GQ90] satisfy these properties. We also show that more recent state-of-the-art protocols in MPC-in-the-head paradigm [IKOS07] like KKW [KKW18] and Ligero [AHIV17] have these properties. We formalize the notion of "$\mathscr{F}$-universally simulatable MPC protocols", which produce stackable Σ-protocols when compiled using MPC-in-the-head [IKOS07]. This formalization is highly non-trivial and requires paying careful attention to the distribution of MPC-in-the-head transcripts. Our key observation is that transcripts generated when executing one circuit can often be seamlessly *reinterpreted* as though they were generated for another circuit (usually of similar size). We refer the reader to Section 4.6 for more details on stackable Σ-protocols.

Stacking Different Σ-Protocols. The compiler presented above allows stacking transcripts for a single Σ-protocol, with a single associated NP language, evaluated over different statements e.g., $(x_1 \in \mathscr{L}) \vee \ldots \vee (x_\ell \in \mathscr{L})$. 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.* $(x_1 \in \mathscr{L}_1) \vee (x_2 \in \mathscr{L}_2) \vee \ldots \vee (x_\ell \in \mathscr{L}_\ell)$.

We start by noting that it is possible to create a "meta-language" to cover multiple languages of interest, and thereby generalize our previous compiler in a straightforward way. For instance, one could create a language $\mathscr{L}$ with an associated relation function that embeds the relation functions for $\mathscr{L}_1, \ldots, \mathscr{L}_\ell$, making $\mathscr{L}$ some form of circuit satisfiability language. A single Σ-protocol could then be used to cover all these languages. Unfortunately, this approach — intuitively equivalent to creating zero-knowledge protocols for all NP complete problems by reducing to a single problem — will often result in high concrete overheads. In rare cases, however, it may be practically efficient; if the languages $\mathscr{L}_1, \ldots, \mathscr{L}_\ell$ are all circuit satisfiability for circuits with the same multiplicative complexity, finding an efficient representation might be easy.

This "meta-language" approach still requires the use of a single Σ-protocol. It would be preferable to allow "cross-stacking," or using different Σ-protocols for each clause in the disjunction.[9] The key impediment to applying our self-stacking compiler to different Σ-protocols is that the distribution of third round messages between two different Σ-protocols may be very different. For example, a statement with three

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

clauses may be composed of one Σ-protocol defined over a large, finite field, another operating over a boolean circuit, and a third that is constructed from elements of a discrete logarithm group. Thus, attempting to use the simulator for one Σ-protocol with respect to the third round message of another might result in a domain error; there may be no set of accepting transcripts for the Σ-protocols that share a third round message. As re-using third round messages is the way we reduce communication complexity, this dissimilarity might appear to be insurmountable.

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

More formally, stacking different Σ-protocols requires an efficient, invertible mapping from each third round message space into some shared distribution $\mathscr{D}$ (*e.g.* random bitstrings in the example above). Intuitively, $\mathscr{D}$ 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 $\mathscr{D}$. Any third round message for one of the Σ-protocols can be mapped into $\mathscr{D}$ 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 $\mathscr{D}$. 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 4.8, we show that this compiler can be efficiently applied to stack many Σ-protocols with each other, including MPC-in-the-head protocols like KKW [KKW18] and Ligero [AHIV17].

In Section 4.9, we briefly explore extended the ideas above to produce zero-knowledge proofs for proofs of partial knowledge, *i.e.* statements where the prover wishes to prove that it has witnesses to at least k out of the ℓ clauses. We note that

solving this problem requires additional structure not present in the purely disjunctive setting. The communication complexity introduced by the compiler we present has an additive overhead that is linear in ℓ, making it less efficient than the other compilers we present in this work. We belive improving on this result is interesting future work.

Paper Organization. The paper is organized as follows: we present required preliminaries Section 4.4 and the interface for partially-binding commitment schemes in Section 4.5. In Section 4.6 we cover the properties of Σ-protocols that our compiler requires and give examples of conforming Σ-protocols. We present our self-stacking compiler in Section 4.7and our cross-stacking compiler in Section 4.8.Finally, in Section 4.9, we give an overview of extending our work to proofs of partial knowledge and in Section 4.10 we discuss the concrete efficiency of instantiating our compilers.

4.4 Preliminaries

Notation

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

Σ-Protocols

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

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

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

– $b \leftarrow \phi(x,a,c,z)$: *On input the statement x, prover's messages a,z and the challenge c, this algorithm run by V, outputs a bit $b \in \{0,1\}$.*

A Σ-protocol has the following properties:

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

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

– ***Special Soundness.*** *A Σ-Protocol $\Pi = (A,Z,\phi)$ is said to have special soundness if there exists a PPT extractor $\mathscr{E}$, 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*

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

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

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

Secure Multiparty Computation

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

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

4.5 Partially-Binding Vector Commitments

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

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

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

The properties satisfied by the above algorithms are as follows:

(Perfect) Hiding: *The commitment key* ck *and commitment* com *(perfectly) hides the binding positions B and the equivocated values, even when opening the commitment. Moreover, the distribution of randomness r computed using* EquivCom *should match $\mathscr{R}$. Formally, for all $\mathbf{v} \in \mathscr{M}^\ell$, $B^{(1)}, B^{(2)} \in \binom{[\ell]}{t}$, a 'valid equivocation' vector $\mathbf{v}' \in \mathscr{M}^\ell$ i.e. $\forall i \in B^{(1)} : \mathbf{v}_i = \mathbf{v}'_i$ and $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, the following distributions are equal:*

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

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

Note that this definition implies (1) that an adversary cannot distinguish between commitment keys ck *generated using different binding sets of the same size, (2) an adversary cannot distinguish between commitments generated using* BindCom *and* EquivCom*, and (3) the opening of the commitment still hides if the opening vector was produced by equivocation.* [11]

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

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

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

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

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

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

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

[11] Note that the definition of hiding has been updated since initial publication.

Relation To Similar Notions

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

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

Somewhere Statistically Binding Commitments. In concurrent work Fauzi, Lipmaa and Pindado [FLPS20] defines a very similar notion of a *somewhere statistically binding* commitment scheme, which provide hiding commitment to vectors and is binding in a subset of the indexes, however their definition/motivation differs in a few important ways: 1) we do not require extraction, as a result partially binding commitments do not imply oblivious transfer unlike SSB commitments, indeed we know of constructions of partially binding commitments from non-blackbox commitments in the random oracle model. 2) we also do not require statistical binding for our compiler: there is a trade-off between statistical/computational binding of the partially binding vector commitment and computational/statistical zero-knowledge respectively of the compiled protocol; the instansiations we provide in this paper choose perfect hiding. 3) for our applications, the party sampling the commitment key generator cannot be trusted. 4) for efficiency we require $|\mathsf{ck}|$ to be small.

Partially-Binding Vector Commitments from Discrete Log

We now present a simple and concretely efficient construction of t-out-of-ℓ partially-binding vector commitments from the discrete log assumption. The idea is to have the committer use a Pedersen commitment for each element in the vector. Recall that a Pedersen commitment to the message $m \in \mathbb{Z}_{|\mathbb{G}|}$ with public parameters $g, h \in \mathbb{G}$ is computed as $g^m h^r$ for a random value r. The binding property of Pedersen commitments relies on the committer not knowing the discrete log of g with respect to h. For our partially-binding vector commitment scheme, the commitment key is a set of public parameters for the Pedersen commitments, constructed such a way that the committer knows discrete logs for exactly $\ell - t$ parameters. This is done by having the committer pick $\ell - t$ of the parameters and computing the remaining t parameters by interpolating in the exponent. More formally, let use begin by fixing some notation. Let $\mathbb{Z}_{|\mathbb{G}|}$ be a prime field. In our construction, we implicitly treat indexes $i \in [0, |\mathbb{G}| - 1]$ as field elements, *i.e.* there is an implicit bijective map between $[0, |\mathbb{G}| - 1]$ and $\mathbb{Z}_{|\mathbb{G}|}$ (e.g. $i \mod |\mathbb{G}| \in \mathbb{Z}/(|\mathbb{G}|)$). Let $\mathscr{X} \subseteq \mathbb{Z}_{|\mathbb{G}|}$ and $j \in \mathscr{X}$, define $L_{(\mathscr{X},j)}(X) := \prod_{m \in \mathscr{X}, m \neq j} \frac{X-m}{j-m} \in \mathbb{Z}_{|\mathbb{G}|}[X]$ i.e. the unique degree $|\mathscr{X}| - 1$ polynomial for which $\forall x \in \mathscr{X} \setminus \{j\} : L_{(\mathscr{X},j)}(x) = 0$ and $L_{(\mathscr{X},j)}(j) = 1$. The formal description

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

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

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

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

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

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

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

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

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

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

Figure 4.2: t-of-ℓ binding commitment from discrete log in the CRS model.

of the commitment scheme can be found in Figure 4.2. While our construction does require a CRS, we note that the CRS is just two randomly selected group elements[12], which in practice can be generated by hashing a 'nothing-up-by-sleeve' constant to the curve by using a cryptographic hash function.

Theorem 1 *Under the discrete log assumption (Definition 13), for any (t, ℓ) with $t < \ell$: the scheme shown in Figure 4.2 is a family of (perfectly hiding, computationally binding) t-of-ℓ partially binding commitment schemes.*

[12] Like regular Pedersen commitments

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

Remark 1 *To commit to longer strings a collision resistant hash* $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_{|\mathbb{G}|}$ *is used to compress each coordinate before committing using* $\mathsf{BindCom}/\mathsf{EquivCom}$ *as a black-box: by committing to* $\mathbf{v}' = (\mathsf{H}(v_1), \ldots, \mathsf{H}(v_\ell))$ *instead. Note that the discrete log assumption (Definition 13), used above, also implies the existance of collision resistant hash functions.*

Definition 5 (Discrete Log Assumption) *There exists a PPT algorithm* $\mathsf{GenGroup}(1^\lambda)$ *which returns a description of a prime-order cyclic group* $\mathbb{G}$ *(written multiplicatively) which admits efficient sampling, st. for all PPT algorithms* $\mathscr{A}$*:*

$$\Pr\left[\mathscr{A}(1^\lambda, \mathbb{G}, g, h) = y \mid \mathbb{G} \leftarrow \mathsf{GenGroup}(1^\lambda); h \xleftarrow{\$} \mathbb{G}; y \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}; g \leftarrow h^y\right] = \mathsf{negl}(\lambda)$$

For some negligible function $\mathsf{negl}(\lambda)$*.*

Proof 1 (Theorem 1) *Completeness of partial equivocation for the scheme in Figure 4.2 is easily seen (follows from equivocation of Pedersen commitments), so we focus on computational binding and perfect hiding.*

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

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

Then the PPT algorithm $\mathscr{A}'$ *shown in Figure 4.10 wins the discrete log game (computing* y_0 *st.* $g_0 = h^{y_0}$*) with probability* $\geq \varepsilon$*. To see this observe that, when* $\mathscr{A}_k$ *wins the binding game: it follows that there exists a set* S *such that its complement* $\overline{S}$ *has size* $|\overline{S}| \geq \ell - t + 1$ *and since* $\forall \alpha, \beta \in [k] : (\mathsf{com}_1, \ldots, \mathsf{com}_\ell) = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}^{(\alpha)}) = \mathsf{BindCom}(\mathsf{pp}, \mathsf{ck}, \mathbf{v}^{(\beta)}) \in \mathbb{G}^\ell$*, we can extract* $y_i \in \mathbb{Z}_{|\mathbb{G}|}$ *st.* $g_i = h^{y_i}$ *whenever* $\mathbf{v}_i^{(\alpha)} \neq \mathbf{v}_i^{(\beta)}$ *by observing:*

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

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

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

1 : Let $\mathsf{pp} = (\mathbb{G}, g_0, h)$

2 : $(\mathsf{ck}, \mathbf{v}^{(1)}, \ldots \mathbf{v}^{(k)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k)}) \leftarrow \mathscr{A}_k(1^\lambda, \mathsf{pp})$

3 : $\overline{S} = \{i \mid \exists\, (\alpha, \beta) : \mathbf{v}_i^{(\alpha)} \neq \mathbf{v}_i^{(\beta)}\} \subseteq [\ell]$, **if** $|\overline{S}| \leq \ell - t$: **return** $\perp$

4 : **for** $i \in \overline{S}$ compute the discrete log in h: $y_i \leftarrow (\mathbf{r}_i^{(\alpha)} - \mathbf{r}_i^{(\beta)})/(\mathbf{v}_i^{(\beta)} - \mathbf{v}_i^{(\alpha)})$

5 : Pick $\mathscr{X} \subseteq_{\ell-t+1} \overline{S}$, compute $y_0 \leftarrow \sum_{i\in\mathscr{X}} y_i \cdot L_{(\mathscr{X},i)}(0)$

6 : **return** y_0

Figure 4.3: Reduction for partially binding commitment scheme to discrete log.

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

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

From a 1-of-2 partial-binding vector commitment scheme, it is easy to obtain a 1-of-2^q binding scheme in which the communication complexity grows linearly in q (i.e. logarithmically in the dimension of the vector), this is achieved by computing a tree of commitments in which the leafs are the entries of the vector being committed to and each internal node is formed by committing to its children: other commitments. There is one commitment key per level and the final commitment is the root of the tree. This means that the binding indexes in the commitment keys encode a path though the tree, leading to a single binding leaf, where as every other path though the tree can

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

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

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

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

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

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

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

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

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

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

Figure 4.4: Generic construction of 1-of-$\ell_A\ell_B$ binding commitment from a 1-of-ℓ_A binding commitment scheme and a 1-of-ℓ_B binding commitment scheme.

be equivocated at some layer. To formalize this, we describe the case of a tree with just two layers, as described below, then apply the transformation iteratively:

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

Proof 2 *The construction works by forming partially binding commitments to partially binding commitments as follows: 1) split the vector $\mathbf{v}$ into ℓ_B chunks $\hat{v}_1, \ldots, \hat{v}_{\ell_B}$ of size ℓ_A each, 2) commit to each chunk individually using PBComm_A with* the same commitment key ck_A*, obtain commitment $\mathbf{v}^{(\mathbf{B})} = (\mathsf{com}_1, \ldots, \mathsf{com}_{\ell_B})$, 3) commit to the commitments $\mathbf{v}^{(\mathbf{B})}$ using PBComm_B and corresponding commitment key ck_B. This scheme is formally described in Figure 4.4. Binding and hiding follows easily from binding and hiding respectively of PBComm_A and PBComm_B.*

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

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

Proof 3 *Apply the transformation from Figure 4.4 q times iteratively to the scheme from Theorem 1 with $\ell = 2$ and $t = 1$. i.e. let the original scheme from Theorem 1 be PBComm_1, compose PBComm_1 with itself to obtain a new 1-of-2^2 binding scheme PBComm_2, then compose PBComm_2 with PBComm_1 to obtain a 1-of-2^3 binding scheme PBComm_3, then compose PBComm_3 with PBComm_1 to obtain a 1-of-2^4 binding scheme PBComm_4, etc. At every step the communication grows by $\mathtt{CC}(\mathsf{PBComm}_1)$, hence the communication of PBComm_q is $q \cdot \mathtt{CC}(\mathsf{PBComm}_1)$.*

4.6 Stackable Σ-Protocols

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

Properties of Stackable Σ-Protocols.

We start by formalizing the definition of a "stackable" Σ-protocol. As discussed in Section 4.3, 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 admissible third round messages, and then constructing the unique appropriate first round message. We refer to such a simulator as an *extended simulator*. A similar notion is considered by Abe *et. al* [AOS02] in their definition of `type-T` signature schemes: a `type-T` signature scheme is essentially the Fiat-Shamir[FS87] heuristic applied to an EHVZK Σ-protocol.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Observation 2 (EHVZK Σ-protocol to challenge-independent EHVZK) *We note that any EHVZK Σ-protocol can be transformed to be challenge-independent EHVZK. Let $\Pi = (A, Z, \phi)$ be an EHVZK Σ-protocol with a 'challenge-dependent' distribution $\mathscr{D}^{(z)}_{x,c}$ over last-round messages and define $\Pi' = (A', Z', \phi')$ as shown below:*

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

1: *Sample* $\Delta \xleftarrow{\$} \{0,1\}^{\kappa}$
2: *Run* $a \leftarrow A(x, w; r^{\mathsf{P}})$
3: **return** (a, Δ)

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

1: *Define* $c = c' \oplus \Delta$
2: *Run* $z \leftarrow Z(x, w, c; r^{\mathsf{P}})$
3: **return** (z, c)

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

1: *Parse* $a' = (a, \Delta), z' = (z, \_)$
2: *Define* $c = c' \oplus \Delta$
3: **return** $\phi(x, w, c, z)$

$\mathscr{S}^{\text{CIEHVZK}'}(x, c', z') \mapsto a'$

1: *Parse* $z' = (z, c)$
2: *Define* $\Delta = c \oplus c'$
3: *Run* $a \leftarrow \mathscr{S}^{\text{EHVZK}}(x, c, z)$
4: **return** (a, Δ)

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

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

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

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

Re-use Property: Recyclable Third Round Messages.

The next property that our stacking compilers require is that the distribution of third round messages does not significantly rely on the statement. In more detail, given a fixed challenge, the distribution of possible third round messages for any pair of statements in the language are indistinguishable from each other. We formalize this

property by using $\mathscr{D}_c^{(z)}$ 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 $\mathscr{D}_c^{(z)}$. We now formally define this property:

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

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

This property is fundamental to stacking, as it means that the contents of the third round message do not 'leak information' about the statement used to generate the message. This means that the message can be safely re-used to generate transcripts for the non-active clauses and an adversary cannot detect which clause is active.[13] 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 9 (Stackable Σ-Protocol) *We say that a Σ-protocol $\Sigma = (A, Z, \phi)$ is* stackable*, if it is EHVZK (see Definition 6) and has recyclable third messages (see Definition 8).*

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

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

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

Looking ahead, these observations will be critical in proving security of our compilers in Sections 4.7 and 4.8.

[13] We further elaborate on this in Remark 2.

Classical Examples of Stackable Σ-Protocols

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

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

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

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

Proof 4 *To see that Π_ψ is stackable, define an extended simulator and check for recyclable third messages:*

1. *Π_ψ is EHVZK: Let $\mathscr{D}_{x,c}^{(z)} \coloneqq \{z \mid z \xleftarrow{\$} \mathfrak{G}_1^*\}$, let $\mathscr{S}^{\text{EHVZK}}(1^\lambda, x, c, z) \coloneqq \psi(z) *_2 x^{-c}$*
2. *Π_ψ has recyclable third messages: Observe that $\forall x_1, x_2 : \mathscr{D}_{x_1,c}^{(z)} = \mathscr{D}_{x_2,c}^{(z)} = \mathscr{U}(\mathfrak{G}_1^*)$*[14].

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

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

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

[14] Uniform distribution over $\mathfrak{G}_1^*$.

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

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

We also show that a variant of Blum's classic 3 move protocol [Blu87] for graph Hamiltonicity is stackable in Section 4.11. This is not surprising, since in the third round the prover either: (1) opens a Hamiltonian path in a permutation of the graph. (2) provides the randomness for the commitment to the permuted adjacency matrix. In either case the distribution of third round messages only depends on the number of vertices in the Hamiltonian graph: either a cycle of n vertices or n^2 openings of commitments (random strings).

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

The proof of Lemma 2 can be found in Section 4.11

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 $\mathscr{R}(x, \cdot)$ using the witness w as input of the parties, and commits to the views of each party. An honest verifier then selects a random subset of the views to be opened and verifies that those views are consistent with each other and with an honest execution, where the output of Π is 1.

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

Achieving recyclable third messages. To show that these Σ-protocols have recyclable third messages, we observe that in many MPC protocols, an *adversary's view can often be condensed and decoupled from the structure of the functionality/circuit* being

evaluated. We elaborate this point with the help of an example protocol — semi-honest BGW [BGW88].

Recall that in the BGW protocol, parties evaluate the circuit in a gate-by-gate fashion on secret shared inputs[15] 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 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.

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

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

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

IKOS Compiler. Let $f = \mathscr{R}(x, \cdot)$. In the first round, the prover runs ExecuteMPC on f and the witness w to obtain views of the parties and commits to each of these views. In the second round, the verifier samples a random subset of parties as its challenge message. Size of this subset is equal to the maximal corruption threshold of the MPC protocol. In the third round, the prover uses CondenseViews to obtain condensed views for this subset of parties and sends them to the verifier along with the randomness used to commit to the original views of these parties in the first round. The verifier runs ExpandViews on f and the condensed views received in the third round to obtain the corresponding original views. It checks if these are consistent with each other and are valid openings to commitments sent in the first round. Depending on the corruption threshold and the security achieved by the underlying MPC protocol, the above steps might be repeated a number of times to reduce the soundness error. Below we restate the main theorem from [IKOS07], which also trivially holds for our modified variant.

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

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 $\mathscr{F}$, and require the MPC protocol to be such that the condensed views can be expanded for any $f \in \mathscr{F}$. For our purposes, it would suffice, even if the condensed view of the adversary is dependent on the final output of the protocol, as long as it is independent of the functionality. This is because, in our context, the circuit being evaluated will be a relation circuit with the statement hard-coded and should always output 1 in order to convince the verifier.

Definition 10 ($\mathscr{F}$-universally simulatable MPC) *Let Π be an n-party MPC protocol that is capable of securely computing any function $f \in \mathscr{F}$ (where $\mathscr{F} : \mathscr{X}^n \to \mathscr{O}$) against any semi-honest adversary $\mathscr{A}$ who corrupts a set $\mathscr{I} \subset [n]$ of parties, such that $\mathscr{I} \in \mathscr{C}$, where $\mathscr{C}$ is the set of admissible corruption sets. We say that Π is $\mathscr{F}$-universally simulatable if there exists a 3-tuple of PPT functions* (ExecuteMPC, ExpandViews, CondenseViews) *and a non-uniform PPT simulator $\mathscr{S}^{\text{F-MPC}} : \mathscr{F} \times \mathscr{C} \times \mathscr{O} \to V^*$, 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 $\{x_i\}_{i\in[n]} \in \mathscr{X}^n$ and a function $f \in \mathscr{F}$ as input and returns the views $\{\mathsf{view}_i\}_{i\in[n]}$ of all parties and their output $o \in \mathscr{O}$ in protocol Π.*
- $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}} \leftarrow \mathsf{CondenseViews}(f, \mathscr{I}, \{\mathsf{view}_i\}_{i\in\mathscr{I}}, o)$*: This function takes as input the set of corrupt parties $\mathscr{I} \in \mathscr{C}$, views of the corrupt parties $\{\mathsf{view}_i\}_{i\in\mathscr{I}}$ and the output of the protocol $o \in \mathscr{O}$ and returns their condensed views $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}}$.*
- $\{\mathsf{view}_i\}_{i\in\mathscr{I}} \leftarrow \mathsf{ExpandViews}(f, \mathscr{I}, \{\mathsf{con.view}_i\}_{i\in\mathscr{I}}, o)$*: This function takes as input the functionality $f \in \mathscr{F}$, set of corrupt parties $\mathscr{I} \in \mathscr{C}$, condensed views $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}}$ of the corrupt parties and the output of the protocol $o \in \mathscr{O}$ and returns their views $\{\mathsf{view}_i\}_{i\in\mathscr{I}}$.*
- $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}} \leftarrow \mathscr{S}^{\text{F-MPC}}(f, \mathscr{I}, \{x_i\}_{i\in\mathscr{I}}, o)$*: The simulator takes as input the functionality $f \in \mathscr{F}$, set of corrupt parties $\mathscr{I} \in \mathscr{C}$, inputs of the corrupt parties $\{x_i\}_{i\in\mathscr{I}} \in \mathscr{X}^{|\mathscr{I}|}$ and the output of the protocol $o \in \mathscr{O}$ and returns simulated condensed views $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}}$ of the corrupt parties.*

And these functions satisfy the following properties:

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

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

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

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

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

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

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

Remark 4 (Topology Decoupled Garbled Circuits and $\mathscr{F}$-universally simulatable MPC.) *The notion of topology decoupled garbled circuits introduced by Kolesnikov [Kol18] is a special case of $\mathscr{F}$-universally simulatable MPC: a topology decoupled garbled circuit (E,T) separates the cryptographic material (E, e.g. garbling tables) and topology (T, i.e. wiring) of a garbled circuit and (informally stated) requires that generating E for different topologies introduces indistinguishable distributions. Letting X be the garbled input labels[16] held by the evaluator, in $\mathscr{F}$-universally simulatable terminology (E,X) would constitute the "condensed view", while (E,X,T)[17] would constitute the "expanded views" , indistinguishablilty of simulated views for functions with the same number of gates and inputs follows easily from the "topology decoupling" of the garbled circuits and the uniform distribution of the input labels.*

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

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

Proof 5 *We define the distribution $\mathscr{D}^{(z)}_{x,c}$, where $c \in \{0,1\}^\kappa$ describes a set of players $\mathscr{I} \in \mathscr{C}$ as follows:*

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

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

[16] Obtained using an oblivious transfer.

[17] Where T can be computed from f.

It runs $\mathsf{ExpandViews}(f, \mathscr{I}, \{\mathsf{con.view}_i\}_{i\in\mathscr{I}}, 1)$ *to obtain original views* $\{\mathsf{view}_i\}_{i\in\mathscr{I}}$. *It then commits to these original views of the opened parties* $\{\mathsf{com}_i = \mathsf{Com}(\mathsf{view}_i; r_i)\}_{i\in\mathscr{I}}$, *and generates dummy commitments* $\{\mathsf{com}_i = \mathsf{Com}(0; r_i)\}_{i\in[n]\setminus\mathscr{I}}$ *for the views of the remaining parties, using some additional randomness* $\{r_i\}_{i\in[n]\setminus\mathscr{I}}$. *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 $\mathscr{H}_0$ *be the distribution over a real transcript and* $\mathscr{H}_3$ *be the above simulated transcript. We define the following intermediate hybrids:*

- $\mathscr{H}_1$*: Compute dummy commitments instead of honest ones for the unopened players in the first round. Indistinguishability between* $\mathscr{H}_0$ *and* $\mathscr{H}_1$ *follows from the hiding property of commitments.*

- $\mathscr{H}_2$*: Instead of honestly computing* $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}}$, *sample these condensed views from* $\mathscr{S}^{\text{F-MPC}}(f, \mathscr{I}, \{x_i\}_{i\in\mathscr{I}}, 1)$ *and use* $\mathsf{ExpandViews}(f, \mathscr{I}, \{\mathsf{con.view}_i\}_{i\in\mathscr{I}}, 1)$ *to obtain original views* $\{\mathsf{view}_i\}_{i\in\mathscr{I}}$. *Indistinguishability between* $\mathscr{H}_2$ *and* $\mathscr{H}_3$ *follows from indistinguishability of simulated views from real execution (See Definition 10) of the MPC protocol.*

- $\mathscr{H}_3$*: Instead of sampling* $\{\mathsf{con.view}_i\}_{i\in\mathscr{I}}$ *from* $\mathscr{D}_{\{x_i\}_{i\in\mathscr{I}},1}$, *sample these condensed views from* $\mathscr{S}^{\text{F-MPC}}(f, \mathscr{I}, \{x_i\}_{i\in\mathscr{I}}, 1)$. *Indistinguishability between* $\mathscr{H}_2$ *and* $\mathscr{H}_3$ *follows from indistinguishability of simulated views for all functions (See Definition 10) of the MPC protocol and from the fact that condensing-expanding views is a deterministic process.*

From transitivity of computational indistinguishability, it follows that $\mathscr{H}_0 \approx \mathscr{H}_3$. *Hence, this* Σ*-protocol achieves EHVZK. For recyclable third messages, we observe that since* $\mathscr{D}^{(z)}_{x,c}$ *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,* $\mathscr{D}^{(z)}_{c}$ *is the same as* $\mathscr{D}^{(z)}_{x,c}$ *and this protocol is indeed stackable.*

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

We prove the following Lemmas (and give a description of the underlying MPC protocol in KKW and Ligero) in Section 4.14 and Section 4.15 respectively.

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

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

Well-Behaved Simulators

As outlined in Section 4.3, a critical step of our compilation framework is applying the simulator of the underlying Σ-protocols to the inactive clauses. Note that these inactive clauses might not be true (if the language is non-trivial), even though the disjunction is satisfied, as such our framework should be applicable to the case where some of the instances are false.

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

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

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

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

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

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

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

We formally prove the following theorem in Section 4.12.

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

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

4.7 Self-Stacking: Disjunctions With The Same Protocol

We now present a self-stacking compiler for Σ-protocols, presented in Figure 4.5. By self-stacking, we mean a compiler that takes a *stackable* Σ-protocol Π for a language $\mathscr{L}$ and produces a Σ-protocol for language with disjunctive statements of the form $(x_1 \in \mathscr{L}) \vee (x_2 \in \mathscr{L}) \vee \ldots \vee (x_\ell \in \mathscr{L})$ with communication complexity proportional to the size of a single run of the underlying Σ-protocol (along with an additive factor that is linear in ℓ and λ). The key ingredient in our compiler is the partially-binding vector commitments (See Definition 4), which will allow the prover to efficiently compute verifying transcripts for the inactive clauses.

The compiler generates an accepting transcript (a_α, c, z^*) to the active clause $\alpha \in [\ell]$ using the witness, and then simulates accepting transcripts for each non-active clause, using the extended simulator. Recall that this extended simulator takes in a third round message z and a challenge c and produces a first round message a such that $\phi(x,a,c,z) = 1$. Thus, the prover can *re-use* the third round message z^*, for each simulated transcript, thereby reducing communication to the size of a single third round message. For a more detailed overview, we refer the reader to Section 4.3.

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

Theorem 5 (Self-Stacking) *Let $\Pi = (A, Z, \phi)$ be a stackable (See Definition 9) Σ-protocol for the NP relation $\mathscr{R} : \mathscr{X} \times \mathscr{W} \to \{0,1\}$ and let* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv}, \mathsf{BindCom})$ *be a 1-out-of-ℓ binding vector commitment scheme (See Definition 4). For any* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, the compiled protocol $\Pi' = (A', Z', \phi')$ described in Figure 4.5 is a stackable Σ-protocol for the relation $\mathscr{R}' : \mathscr{X}^\ell \times ([\ell] \times \mathscr{W}) \to \{0,1\}$, where $\mathscr{R}'((x_1, \ldots, x_\ell), (\alpha, w)) := \mathscr{R}(x_\alpha, w)$.*

Proof 6 *We now prove that the protocol $\Pi' = (A', Z', \phi')$ described in Figure 4.5 is a stackable Σ-protocol for the relation $\mathscr{R}'((x_1,\ldots,x_\ell),(\alpha,w)) := \mathscr{R}(x_\alpha, w)$.*

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

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

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

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

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

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

Figure 4.5: A compiler for stacking multiple instances of a Σ-protocol.

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

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

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 $\mathscr{L}_1$ and $\mathscr{L}_2$ 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 $(x_1\in\mathscr{L}_1)\vee(x_2\in\mathscr{L}_2)\vee\ldots\vee(x_\ell\in\mathscr{L}_\ell)$. By "Σ-protocols based on a particular technique", we mean (for instance) protocols based on [KKW18]. This can be done by working with a "meta-language" that covers all languages of interest and is supported by that technique. This could for example be an NP complete language, which would allow us to use our self-stacking compiler by first reducing each x_i to an instance of the NP-complete language. However, reducing to NP complete languages without care will often add the significant cost of performing an NP-reduction to the complexity of our compiler, which may no longer be efficient. In many cases, it is often possible to find the "most suitable" meta-language without compromising on the efficiency. For instance, for any MPC-in-the-head style Σ-protocol, this meta-language is as simple as circuit satisfiability for circuits of a given size (where this size is determined based on the language with the largest relation circuit). This can be easily achieved with the help of padding, without incurring any additional overhead. This observation combined with the fact that simulation is deterministic in both KKW and Ligero, we get a protocol for general disjunctions, where the communication is

the same as the communication for a single instance for the clause with the largest relation circuit and additive factor that depends on $\log(\ell)$ and the security parameter.

4.8 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 $(x_1 \in \mathscr{L}_1) \vee (x_2 \in \mathscr{L}_2) \vee \ldots \vee (x_\ell \in \mathscr{L}_\ell)$. 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 4.7, where the Σ-protocols all shared the same techniques, using the meta-language approach. However, that idea crucially relied on the fact that there exists such a meta-language that is also supported by the Σ-protocol technique that we want to stack. To avoid this requirement, we now consider the more complex case where the Σ-protocols rely on different techniques.[18] 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*.

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 $\mathscr{D}_c^{(z)}$. 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 4.7 we used the "meta-language" approach for KKW-based [KKW18] Σ-protocols. We now consider another example using Schnorr-like protocols that does not require us to work with a meta-language:

Example 1 (Preimages of homomorphisms with the same domain [CD98]) *Recall the protocol Π_ψ of Cramer and Damgård described earlier. Any two instances of Π_{ψ_1} and Π_{ψ_2} for one-way homomorphisms $\psi_1 : \mathfrak{G}_1^* \to \mathfrak{G}_2^*$ and $\psi_2 : \mathfrak{G}_1^* \to \mathfrak{G}_3^*$ with the same domain $\mathfrak{G}_1^*$ can be stacked* as through *they were the same protocol using the self-stacking compiler: recall that for both Π_{ψ_1} and Π_{ψ_2}, $\mathscr{D}_c^{(z)}$ is the uniform distribution over $\mathfrak{G}_1^*$. Concrete examples include generalizations of the Chaum-Pedersen [CP93] Σ-protocol $\Pi_{DlogEq,\ell}$ (shown in Figure 4.6), for showing equality of discrete log: for any $(\ell, g_1, \ldots, g_\ell)$ the homomorphism $\psi_{g_1,\ldots,g_\ell} : \mathbb{Z}_{|\mathbb{G}|} \to \mathbb{G}^\ell$ defined as $\psi_{g_1,\ldots,g_\ell}(w) \coloneqq (g_1^w, \ldots, g_\ell^w)$, has the same domain $\mathbb{Z}_{|\mathbb{G}|}$ (different ranges).*

[18] 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 4.7 to stacking Σ-protocols based on differing techniques naturally leads to the question of how well transcripts with differing structures and distributions can be re-used. We highlight these questions in this section under the name cross-stacking.

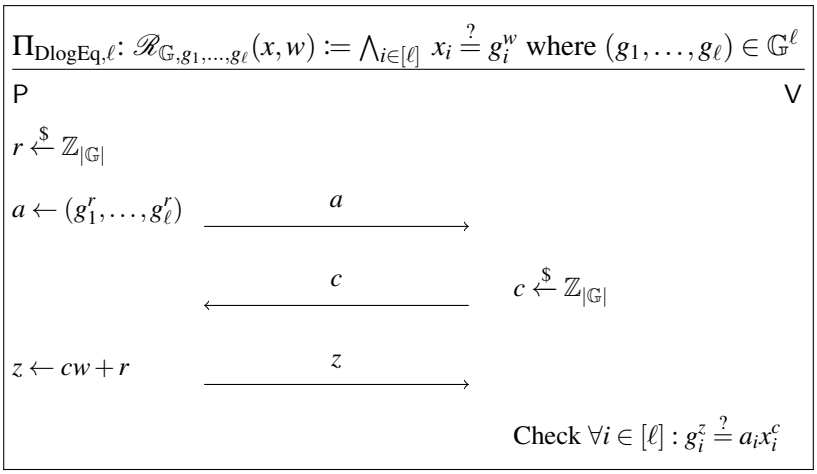

Figure 4.6: Generalized Chaum-Pedersen

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

Example 2 (KKW over different commutative rings) *Consider two KKW-based Σ-protocols. Π_1 is for a language with a relation circuit defined over the ring $\mathbb{F}_{2^k}$, while Π_2 is for a language with relation circuit over $\mathbb{Z}_{2^k}$. If elements of both $\mathbb{F}_{2^k}$ and $\mathbb{Z}_{2^k}$ are encoded as k-bit strings and the multiplicative complexity of the relation circuits are the same, the the bit-wise distribution of $\mathscr{D}_c^{(z)}$ for Π_1 and Π_2 is the same (see Figure 4.14). Therefore, Π_1 and Π_2 can be stacked* as though *they were the same protocol using the self-stacking compiler and their extended simulators will re-use the bit-wise encodings of elements of one ring as though they were bit-wise encodings of the other ring. This approach can be generalized to any pair of finite commutative rings 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 $\mathscr{D}_c^{(z)}$. 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" $\mathscr{D}$ 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. $\mathscr{D}$ 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 $\mathscr{D}$ using a (possibly randomized) embedding function $F_{\Sigma\to\mathscr{D}}$ and a deterministic extraction function $\mathsf{TExt}_{\mathscr{D}\to\Pi}$. For example, $F_{\Pi\to\mathscr{D}}$ may add randomly sampled bits or cryptographic material to a third round message z in order to create an element $d\in\mathscr{D}$. $\mathsf{TExt}_{\mathscr{D}\to\Pi}$ might simply "select" the appropriate bits from d to construct z. We note that $\mathscr{D}$ is independent of c and therefore needs to cover all possible values of c. Thus, if $\mathscr{D}_c^{(z)}$ varies wildly across c for one of the Σ-protocols, $\mathscr{D}$ will need to be large. This, however, is not common in practice; for example, $\mathscr{D}_c^{(z)}$ is the same across all values of c for both KKW and Ligero. We now present the property that we will require for cross-stacking:

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

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

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

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

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

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

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

This property guarantees that third round messages in a Σ-protocol can be embedded into the (possibly larger) distribution $\mathscr{D}$, a generalization of Definition 8. Note that every stackable Σ-protocol is cross simulatable with its own third round message distribution. To make this property useful, we will require that a *set of* Σ-protocols

are all cross simulatable with the *same* distribution $\mathscr{D}$. This property can be trivially satisfied by simply appending the distributions of the underlying stackable Σ-protocols, making $\mathscr{D}$ a tuple of elements of the underlying distributions; the challenge is to find small $\mathscr{D}$ 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 $\mathscr{D}$ that is very similar in size to the distributions over third round messages of these protocols using the example of KKW [KKW18] and Ligero [AHIV17]. This is despite the very distinct features of the two techniques: Ligero has negligible soundness error, players equal to the square-root of the multiplicative complexity of the circuit, and requires a sufficiently large field. KKW, on the other hand, has constant soundness that must be amplified, a constant number of players (independent of the circuit size), and operates over any commutative ring.

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

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

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

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 $\mathscr{D}$; 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, $\mathscr{D}$ contains the maximum number of commitments and pseudorandom bits from between the two protocols.*

Mapping into $\mathscr{D}$ involves determining the size of the padding: if more commitments must be added, the mapping function samples arbitrary values ε and commits to them honestly. Note that these commitments will never *be opened, so the contents do not matter. If more pseudorandom bits are required, the mapping function samples the*

*required number of bits. Extracting a third round function involves selecting the appropriate number of commitments and pseudorandom bits and parsing these bits as needed. Note that if the sizes of C_1 and C_2 are appropriate ($\sqrt{C_1} \approx C_2 \times$(*number of repetitions*)), very little padding will be needed.*

Cross-Stacking 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 $\mathscr{D}$.

Theorem 6 (Cross-Stacking) *Let $\mathscr{D}$ be a distribution. For each $i \in [\ell]$, let $\Pi_i = (A_i, Z_i, \phi_i)$ be a stackable (See Definition 9) Σ-protocol for the NP relation $\mathscr{R}_i : \mathscr{X}_i \times \mathscr{W}_i \to \{0,1\}$, that is cross simulatable w.r.t. to a distribution $\mathscr{D}$, and let $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{EquivCom}, \mathsf{Equiv})$ be a 1-out-of-ℓ binding vector commitment scheme (See Definition 4). For any $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, the protocol $\Pi' = (A', Z', \phi')$ described in Figure 4.7 is a <u>stackable</u> Σ-protocol for the relation $\mathscr{R}'((x_1, \ldots, x_\ell), (\alpha, w)) := \mathscr{R}_\alpha(x_\alpha, w)$.*

We give a proof of this theorem in Section 4.13.

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 $\mathscr{D}$ (let this be $|x_{\mathscr{D}}|$). Thus, the communication complexity of Π' is $(|x_{\mathscr{D}}| + \ell O(\lambda) + |\mathsf{com}| + |r'|)$. 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 $(|x_{\mathscr{D}}| + 2\log(\ell) O(\lambda) + |\mathsf{com}| + |r'|)$, where $O(\lambda)$ can be minimized by using efficient constructions of partially-binding vector commitments, as shown in the complexity analysis of self-stacking.

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

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

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

1. Prover samples a k-universal hash function $H \xleftarrow{\$} \mathscr{H}$ subject to $|\{H(\alpha)\}_{\alpha \in \mathscr{A}}| = k$.

2. Prover commits to the function: $\mathsf{com} \leftarrow \mathsf{Commit}(H; r)$.

3. The prover sends $\mathsf{com}, t_1 = H(\alpha_1), \ldots, t_k = H(\alpha_k)$ to the verifier and for every $i \in [k]$ proves:

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

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

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

The primary challenge in the compiler above is to instantiate Commit and $\mathscr{H}$ such that $\mathsf{com} = \mathsf{Commit}(H; r) \wedge t_i = H(i)$ can be efficiently proven using a Sigma protocol. We instantiate Commit using a scheme which enables commitment to polynomials of degree $k-1$ (a family of k-universal hash functions), note that we do not require a 'polynomial commitment scheme' as commonly defined: in particular

we do not require that $|\mathsf{com}|$ is succinct or the opening proofs have poly-logarithmic time/communication in the degree of the polynomial. The scheme is a natural one based on a linearly homomorphic commitment scheme:

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

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

4.10 Measuring Concrete Efficiency

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

Self-Stacking KKW [KKW18]. Our first measurement that demonstrates the concrete efficiency of our compiler is self-stacking KKW [KKW18]. We compare the results of this protocol to the naïve approach of simply applying CDS [CDS94] to the equivalent disjunctive statement. Specifically, we compare our compiler and CDS applied applied to circuits containing 1000 and 100,000 multiplication gates and sweep between 1 and 1000 clauses. The results of this comparison can be found in Figure 4.8.

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

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

where $\mathsf{Size\text{-}KKW}$ and parameters (M, τ) are derived in KKW [KKW18].

Ring-signatures From Schnorr Signatures [Sch90]. Our second efficiency measurment is constructing ring-signatures [RST01] by applying our compiler to classical

Schnorr Signatures. Specifically, we recursively apply the compiler from Theorem 5 (with the partially binding vector commitments from Figure 4.16), to the Schnorr [Sch90] identification protocol over the Ristretto group of `Curve25519` [Ber06]. Then, we apply the Fiat-Shamir [FS87] heuristic to obtain a signature of knowledge in the random oracle model. We implement the compiled protocol in the Rust programming language; our implementation is open source and is available at `https://github.com/rot256/research-stacksig` .

The concrete size of these ring signatures with 128 bits of security and n parties is $|\sigma| = 64 \cdot \lceil \log_2(n) \rceil + 64$ bytes. We present running times and signatures sizes for these ring signatures in Figure 4.9.[19]

Acknowledgments

We would like to thank the anonymous reviewers of CRYPTO 2021 for their helpful comments on our initial construction of the partially-binding commitments. Additionally, we would like to thank Nicholas Spooner for his helpful comments on the definition of these commitments. Finally, we would like to thank Gennaro Avitabile, Vincenzo Botta, Daniele Friolo and Ivan Visconti, who in their follow-up work "Efficient Proofs of Knowledge for Threshold Relations"[ABFV22] pointed out some subtle definitional issues in the previous version of this paper and constructed an optimized threshold version of our compiler. This motivated us to observe a different (and improved) variant of our threshold construction, which we describe in this updated version. Additionally, we have updated the definition of hiding based on their observations.

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

Definition 13 (Discrete Log Assumption) *There exists a PPT algorithm* $\mathsf{GenGroup}(1^\lambda)$ *which returns a description of a prime-order cyclic group* $\mathbb{G}$ *(written multiplicatively)*

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

which admits efficient sampling, st. for all PPT algorithms $\mathscr{A}$:

$$\Pr\left[\mathscr{A}(1^\lambda,\mathbb{G},g,h)=y \mid \mathbb{G}\leftarrow \mathsf{GenGroup}(1^\lambda); h\xleftarrow{\$}\mathbb{G}; y\xleftarrow{\$}\mathbb{Z}_{|\mathbb{G}|}; g\leftarrow h^y\right]=\mathsf{negl}(\lambda)$$

For some negligible function $\mathsf{negl}(\lambda)$.

Proof 7 (Theorem 1) *Completeness of partial equivocation for the scheme in Figure 4.2 is easily seen (follows from equivocation of Pedersen commitments), so we focus on computational binding and perfect hiding.*

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

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

Then the PPT algorithm $\mathscr{A}'$ shown in Figure 4.10 wins the discrete log game (computing y_0 st. $g_0=h^{y_0}$) with probability $\geq\varepsilon$. To see this observe that, when $\mathscr{A}_k$ wins the binding game: it follows that there exists a set S such that its complement $\overline{S}$ has size $|\overline{S}|\geq\ell-t+1$ and since $\forall\alpha,\beta\in[k]:(\mathsf{com}_1,\ldots,\mathsf{com}_\ell)=\mathsf{BindCom}(\mathsf{pp},\mathsf{ck},\mathbf{v}^{(\alpha)})=\mathsf{BindCom}(\mathsf{pp},\mathsf{ck},\mathbf{v}^{(\beta)})\in\mathbb{G}^\ell$, we can extract $y_i\in\mathbb{Z}_{|\mathbb{G}|}$ st. $g_i=h^{y_i}$ whenever $\mathbf{v}_i^{(\alpha)}\neq\mathbf{v}_i^{(\beta)}$ by observing:

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

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

Perfect Hiding *Recall that we denote the set of binding indexes as B, and its complement (the set of indexes that support equivocation) as E. Observe that for any E the distribution of $\mathsf{ck}=(g_1,\ldots,g_{[\ell]-t})$ is uniform in $\mathbb{G}^{\ell-t}$: since the distribution of $\{g_j\}_{j\in E}$ is uniform and $\{g_j\}_{j\in[\ell-t]}$ is computed as a bijection of $\{g_j\}_{j\in E}$. Hence the distribution of* ck *is independent of E (and B), and the*

binding indexes are perfectly hidden. The perfect hiding of the commitment $(\mathsf{com}_1,\ldots,\mathsf{com}_\ell)$ *follows directly from perfect hiding of Pedersen commitments: each* com_i *is sampled i.i.d. uniform from* $\mathbb{G}$*. Finally, note that r is distributed uniformly in* $\mathbb{Z}^{\ell}_{|\mathbb{G}|}$*, both when committing using* BindCom *and equivocating with* EquivCom.

4.11 Blum87 is Stackable: Proof of Lemma 2

Let $\mathscr{L}^{\text{Ham}}_n \subseteq \{0,1\}^{n\times n}$ be the language of n vertex graphs with a Hamiltonian cycle (represented by adjacency matrices). For any n Blum's classical Σ-protocol for $\mathscr{L}^{\text{Ham}}_n$ is stackable, recall the protocol (Shown in Figure 4.11):

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

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

Proof 8 *(Lemma 2) For* $c \in \{0,1\}$*, define* $\mathscr{D}^{(z)}_c$ *as follows:*

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

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

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

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

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

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

4.12 Well-Behaved Simulators: Proof of Lemma 5

Proof 9 *Given $\Pi = (A, Z, \phi)$, construct the new $\Pi' = (A', Z', \phi')$ with well-behaved a simulator as follows:*

- $A'(x, w; r) := (a, \perp)$ *where* $a \leftarrow A(x, w; r)$
- $Z'(x, w, c; r) := z$ *where* $z \leftarrow Z(x, w, c; r)$
- $\phi'(x, a', c, z) :=$
  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 Π' the prover can either choose to attempt guessing the challenge c (sending $a' = (\perp, c)$), or, he can run the original protocol (sending $a' = (a, \perp)$). The (well-behaved) simulator $\mathscr{S}'$ of Π' first runs the simulator $\mathscr{S}$ of Π, if the simulated transcript (a, c, z) is accepting then output the transcript, otherwise $\mathscr{S}'$ 'guesses' the challenge:

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

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

- *Completeness: follows from completeness of Π. In particular in the real executions $a' = (a, \perp)$ always.*
- *Special Honest Verifier Zero-Knowledge: For every $x \in \mathscr{L}$ and $c \in \{0,1\}^\lambda$, the output of the original simulator $(a, z) \leftarrow \mathscr{S}(1^\lambda, x, c)$ must always be accepting $\phi(a, c, z) = 1$ by SHVZK of Π. Hence the distribution of $\mathscr{S}'$ on statements $x \in \mathscr{L}$ is Since the distribution in the real execution will always have $a' = (a, \perp)$*
- *Special Soundness: Suppose we get two transcripts with a shared first-round message: (a', c_1, z_1, c_2, z_2) st. $\phi'(a', c_1, z_1) = 1$, $\phi(a', c_2, z_2) = 1$ and $c'_1 = c'_2$. Consider the two distinct forms that a' can take:*
  1. *When $a' = (\perp, c)$ then clearly there does not exists two accepting transcripts with different challenges c_1 and c_2 since $c = c_1 = c_2$. Hence the assumption that $a' = (\perp, c)$ is a contradiction.*
  2. *When $a' = (a, \perp)$ then z_1, z_2 must satisy $\phi(a, c_1, z_1) = 1$ and $\phi(a, c_2, z_2) =$ 1. Therefore, we can extract a witness $w \leftarrow \mathscr{E}(a, c_1, z_1, c_2, z_2)$ using the extractor of Π.*

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

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

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

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

We now prove that the protocol $\Pi' = (A', Z', \phi')$ described in Figure 4.7 is a stackable Σ-protocol for the relation $\mathscr{R}'((x_1, \ldots, x_\ell), (\alpha, w)) := \mathscr{R}_i(x_\alpha, w)$.

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

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

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

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

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

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

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

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

Therefore Σ' is a stackable Σ-protocol.

4.14 Overview of [KKW18] and Proof of Lemma 3

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

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

- $\texttt{Input}(\gamma)$: assigns to the next R-element from the witness to the wire w_γ.

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

Underlying MPC. It is most clearly seen how preprocessing as used in KKW fits into our framework by simply viewing the MPC as an $n+1$ player protocol, where a 'preprocessing player' P_0 acts as a dealer (shown in Figure 4.12) and sends the 'online players' $P_1,\ldots,P_n$ correlated randomness via point-to-point channels. The MPC is passively secure against the corruption patterns $\mathscr{C} = \{\{0\} \cup \binom{[n]}{n-1}\}$ i.e. the 'preprocessing player' or any $n-1$ subset of the 'online players'. During the online phase of KKW (shown in Figure 4.13) the n players hold additive shares $[\lambda_\gamma]^{(i)}$ of masks $\lambda_\gamma = \sum_{i=1}^{n} [\lambda_\gamma]^{(i)}$ and public maskings $z_\gamma = v_\gamma - \lambda_\gamma$ of the value v_γ assigned to the w_γ, i.e. the value of w_γ is $v_\gamma = z_\gamma + \sum_{i=1}^{n} [\lambda_\gamma]^{(i)}$.

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

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

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

1. All broadcast messages not sent by players in $\{P_i\}_{i\in\mathscr{I}}$:
   a) The masked input wires z_γ for gates $\texttt{Input}(\gamma)$.
   b) The $[s_\gamma]^{(p)}$ shares sent by player $P_p, p \notin \mathscr{I}$ during multiplication.

2. The corrections Δ sent by player 0.

3. The $n-1$ individual per-player PRG seeds $\{s_i\}_{i\in\mathscr{I}}$,

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

Remark 5 *We do not need to include in $\mathscr{T}$ 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 $\{[\lambda_\alpha]^{(i)}\}_{i\in\mathscr{I}}$ as: $[\lambda_\alpha]^{(p)} = o - z_\alpha - \sum_{i\in\mathscr{I}}[\lambda_\alpha]^{(i)}$.*

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.

[KKW18] is Stackable: Proof of Lemma 3

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

Proof 10 *(Lemma 3) Let $\mathscr{D}^{(\texttt{real})}$ be the real distribution over condensed views for a particular $\mathscr{I}$. The simulator is given in Figure 4.14. Consider the two cases:*

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

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

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

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

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

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

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

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

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

4.15 Overview of Ligero and Proof of Lemma 4

In this section, we discuss the MPC model used in Ligero, give an overview about why their underlying MPC is $\mathscr{F}$-universally simulatable , recall the construction of their MPC protocol and finally give a formal proof for why their protocol is $\mathscr{F}$-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 3), we assume that the random string r is obtained by the sender using a random oracle by providing the list of all the messages that it computes in the first phase as input. Given this slight modification, we observe that the underlying MPC protocol in Ligero is $\mathscr{F}$-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 $\mathscr{F}$-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 $\mathscr{F}$-universally simulatable .

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

Definition 14 (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 15 (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 16 (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 $\ell \le k$. For $u \in L$, we define the message $\mathsf{Dec}_\zeta(u)$ to be $(p_u(\zeta_1),\ldots,p_u(\zeta_\ell))$, where p_u is the polynomial (of degree $< k$) corresponding to u. For $U \in L^m$ with rows $u^1,\ldots,u^m \in L$, we let $\mathsf{Dec}_\zeta(U)$ be the length-$m\ell$ vector $x = (x_{11},\ldots,x_{1\ell},\ldots,x_{m1},\ldots,x_{m\ell})$ such that $(x_{i1,\ldots,x_{i\ell}}) = \mathsf{Dec}_\zeta(u^i)$ for $i \in [m]$. Finally, when ζ is clear from the context, we say that U encodes x if $x = \mathsf{Dec}_\zeta(U)$.*

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

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

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

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

- It then constructs matrix $P_{\mathsf{add}} \in \mathbb{F}^{m\ell \times m\ell}$ such that the j^{th} row of $P_{\mathsf{add}} w$ equals $\beta_a + \beta_b - \beta_c$ where β_a, β_b and β_c correspond to the j^{th} addition gate in w.

- It then samples random codewords $U^w, U^x, U^y, U^z \in L^m$ where $L = \mathsf{RS}_{\mathbb{F},n,k,\eta}$ subject to $w = \mathsf{Dec}_\zeta(U^w), x = \mathsf{Dec}_\zeta(U^x), y = \mathsf{Dec}_\zeta(U^y), z = \mathsf{Dec}_\zeta(U^z)$ where $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ is a sequence of distinct elements disjoint from $(\eta_1, \ldots, \eta_n)$.

- Let $u', u^x, u^y, u^z, u^0, u^{\mathsf{add}}$ be auxiliary rows sampled randomly from L where each of $u^x, u^y, u^z, u^{\mathsf{add}}$ encodes an independently samples random ℓ messages $(\gamma_1, \ldots, \gamma_\ell)$ subject to $\Sigma_{c\in[\ell]}\gamma_c = 0$ and u^0 encodes 0^ℓ.

- It sets $U \in L^{4m}$ as a juxtaposition of the matrices $U^w, U^x, U^y, U^z \in L^m$. It also computes $r^* \leftarrow H^{\mathsf{RO}}(U)$, where $r^* = (r, r^{\mathsf{add}}, r^x, r^y, r^z, r^q)$, such that $r \in \mathbb{F}^{4m}, r^{\mathsf{add}}, r^x, r^y, r^z \in \mathbb{F}^{m\ell}$, $r^q \in \mathbb{F}^m$.

- It sends $U[j], u'[j], u^x[j], u^y[j], u^z[j], u^0[j], u^{\mathsf{add}}[j]$ to server j (for $j \in [n]$), where $U[j]$ represents the j^{th} column in U. It also sends r^* to each server.

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

- Compute and send $v[j] = r^T U[j] + u'[j]$.

- – Compute constructs matrix $P_{\mathsf{add}} \in \mathbb{F}^{m\ell \times m\ell}$ such that the j^{th} row of $P_{\mathsf{add}} w$ equals $\beta_a + \beta_b - \beta_c$ where β_a, β_b and β_c correspond to the j^{th} addition gate in w.[20]
  - 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:
  - $q^x[j] = u^x[j] + \Sigma_{i\in[m]} r_i^x(j) \cdot U^x[i, j] + \Sigma_{i=m+1}^{2m} r_i^x(j) \cdot U^w[i - m, j]$.
  - $q^y[j] = u^y[j] + \Sigma_{i\in[m]} r_i^y(j) \cdot U^y[i, j] + \Sigma_{i=m+1}^{2m} r_i^y(j) \cdot U^w[i - m, j]$.
  - $q^z[j] = u^z[j] + \Sigma_{i\in[m]} r_i^z(j) \cdot U^z[i, j] + \Sigma_{i=m+1}^{2m} r_i^z(j) \cdot U^w[i - m, j]$.
  - $p_0[j] = u^0[j] + \Sigma_{i\in[m]} r^q[i] \cdot (U^x[i, j] \cdot U^y[i, j] - U^z[i, j])$.

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

Ligero is Stackable: Proof of Lemma 4

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

- Sample a random vector $v \in \mathbb{F}$.
- For each $j \in \mathscr{I}$, sample random elements from $\mathbb{F}$ for $U^x[j], U^y[j], U^z[j], U^w[j]$.
- For each $j \in \mathscr{I}$, 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 $\mathscr{F}$-universal simulator.

ExpandViews : We now describe the expand views function for this protocol

- For each $j \in \mathscr{I}$, 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 $\{q^{\mathsf{add}}[j]\}_{j\in\mathscr{I}}$ 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 $\{q^{\mathsf{add}}[j]\}_{j\in[n]\setminus\mathscr{I}}$
- For each $a \in \{x,y,z\}$, use $\{q^a[j]\}_{j\in\mathscr{I}}$ to extrapolate a polynomial q^a of degree $< k+\ell-1$ such that $\Sigma_{c\in[\ell]} q^a(\zeta_c) = 0$ and output $\{q^a[j]\}_{j\in[n]\setminus\mathscr{I}}$.
- Use $\{q^0[j]\}_{j\in\mathscr{I}}$ to extrapolate a polynomial q^0 of degree $< 2k-1$ such that $p_0(\zeta_c) = 0$ for every $c \in [\ell]$ and output $\{q^0[j]\}_{j\in[n]\setminus\mathscr{I}}$.

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

4.16 Partially Binding Vector Commitments in the ROM

In this section, we present our optimized construction of partially binding vector commitments. We show that this construction is secure if the discrete log assumption holds. However, showing a direct reduction is cumbersome. Instead, we first formalize a variant of the discrete log assumption, called AdaptiveDlog that is more convenient for our purposes. We will use this variant, presented in Definition 13, as a stepping-stone in our analysis. Intuitively, there are 3 elements in play when an adversary wants to break the construction: an element in the CRS h, the element it gets to choose g_*, and the element corresponding to the index at which it would like to cheat, say g. In order to break the construction, the adversary would somehow need to uncover a relationship in the discreet logs between these values. Note the order in which these are chosen: first the CRS value is sampled, then the adversary selects the value g_* which is not binding. Finally, the random oracle "samples" the remaining group element. AdaptiveDlog captures this game directly; we begin by showing that it is equivalent to the discreet log assumption.

Lemma 6 (Discrete Log reduces to AdaptiveDlog**.)** *Let $\mathscr{A}$ be an adversary winning the* AdaptiveDlog *game (Figure 4.15) with probability ε, then there exists an expected polynomial-time adversary $\mathscr{A}'$ computing discrete logs (Definition 13) in $\mathbb{G}$ with probability $\geq \varepsilon - \mathsf{negl}(\lambda)$.*

Proof 11 *Consider the following PPT algorithm $\mathscr{A}'$:*

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

1 : *Send h to* $\mathscr{A}$; $\mathscr{A}$ *returns* $g_* \in \mathbb{G}$

// Initial query in the h row

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

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

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

// Probe the h row without replacement.

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

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

7 : *Rewind* $\mathscr{A}$ *to before message 3 (just before sending g)*

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

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

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

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

// Extract the discrete log.

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

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

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

15 : **return** β

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

Lemma 7 *For a cyclic group* $\mathbb{G}$ *wherein discrete log is intractable (Definition 13), let* $\mathsf{P}_{|\mathbb{G}|} : \mathbb{G} \to \mathbb{G}$ *be a cryptographic permutation (modelled as a invertible random oracle) with inverse* $\mathsf{P}^{-1}_{|\mathbb{G}|} : \mathbb{G} \to \mathbb{G}$*. The construction shown in Figure 4.16 is a (computationally binding and perfectly hiding)* $(\ell - 1)$*-of-*ℓ *partially binding vector commitment scheme.*

Proof 12 *The completeness of partial equivocation is easily seen (follows from equivocation of vector Pedersen commitments), so we focus on computational binding and perfect hiding.*

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

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

Then $\mathscr{A}'$ (Figure 4.17) wins the AdaptiveDlog *game with probability $\varepsilon' = \varepsilon/\mathsf{poly}(\lambda) - \mathsf{negl}(\lambda)$. We lower bound the probability that $h^{\hat{x}} g_*^{\hat{y}} g^{\hat{z}} = 1$ and $\hat{x}, \hat{y}, \hat{z} \neq 0$. Start by observing that in the chain: $\forall i \in [2, \ell] : g_{i+1} = \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$, there can be at most one group element g'_* on which the oracle is queried, but which has not been output by $\mathsf{P}_{|\mathbb{G}|}$: there are $\ell - 1$ outputs and ℓ group elements. If all elements in the chain has been output by $\mathsf{P}_{|\mathbb{G}|}$, then define $g'_* = g_1$. Suppose the reduction guesses correctly and $g_* = g'_*$, this occurs with noticeable probability $1/\mathsf{poly}(\lambda)$. Suppose furthermore that $\mathscr{A}$ wins the binding game, in this case we know: $\Pr_\delta[\mathsf{HW}(\mathbf{w}) \geq 3] = 1 - 1/|\mathbb{G}|$, because $\mathsf{HW}(\mathbf{w}^{(1)}) \geq 2, \mathsf{HW}(\mathbf{w}^{(2)}) \geq 2$ and $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}$ have at least one distinct non-zero position each. Note additionally that $\mathbf{g}^{\mathbf{w}} = \mathbf{g}^{\mathbf{w}^{(1)}} \cdot (\mathbf{g}^{\mathbf{w}^{(2)}})^\delta = 1 \cdot 1^\delta = 1$, furthermore:*

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

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

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

Cross-Stacking Compiler

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

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

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

.

Figure 4.7: A compiler for stacking instances of multiple Σ-protocols.

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

Figure 4.8: Concrete communications complexity for disjunctions over Boolean circuits ($R = \mathbb{F}_2$) with different multiplicative complexity, targeting 128-bits of security: $n = 64, M = 631, \tau = 23$. The communication complexity of our work is computed when recursive stacking is applied using the optimized commitment scheme described in Section 4.16.

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

Figure 4.9: Performance of ring signatures for rings of different sizes.
All benchmarks run on a single core of `AMD EPYC 7601` @ 2.2 GHz.

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

1 : Let $\mathsf{pp} = (\mathbb{G}, g_0, h)$

2 : $(\mathsf{ck}, \mathbf{v}^{(1)}, \ldots \mathbf{v}^{(k)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k)}) \leftarrow \mathscr{A}_k(1^\lambda, \mathsf{pp})$

3 : $\overline{S} = \{i \mid \exists\, (\alpha, \beta) : \mathbf{v}_i^{(\alpha)} \neq \mathbf{v}_i^{(\beta)}\} \subseteq [\ell]$, **if** $|\overline{S}| \leq \ell - t$: **return** $\perp$

4 : **for** $i \in \overline{S}$ compute the discrete log in h: $y_i \leftarrow (\mathbf{r}_i^{(\alpha)} - \mathbf{r}_i^{(\beta)})/(\mathbf{v}_i^{(\beta)} - \mathbf{v}_i^{(\alpha)})$

5 : Pick $\mathscr{X} \subseteq_{\ell-t+1} \overline{S}$, compute $y_0 \leftarrow \sum_{i \in \mathscr{X}} y_i \cdot L_{(\mathscr{X},i)}(0)$

6 : **return** y_0

Figure 4.10: Reduction for partially binding commitment scheme to discrete log.

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

P		V
	κ times in parallel	
$\pi \xleftarrow{\$} \mathrm{S}_n$		
$r_{1,1}, \ldots, r_{n,n} \xleftarrow{\$} \{0,1\}^\lambda$ $\forall i, j \in [n]$: $C_{i,j} \leftarrow \mathsf{Commit}(x_{\pi(i),\pi(j)}; r_{\pi(i),\pi(j)})$		
	$a = (C_{1,1}, \ldots, C_{n,n})$ $\longrightarrow$	
	$\longleftarrow$ c	$c \xleftarrow{\$} \{0,1\}$
$\tau \leftarrow \pi \circ w$ $R_0 \leftarrow (\pi, r_{1,1}, \ldots, r_{n,n})$ $R_1 \leftarrow (\tau, r_{1,\tau(1)}, \ldots, r_{n,\tau(n)}, C_{1,1}, \ldots, C_{n,n})$		
	R_c $\longrightarrow$	
		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: $a \stackrel{?}{=} (C_{1,1}, \ldots, C_{n,n})$

Figure 4.11: 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.

Player 0 computes correlated Beaver triples:

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

Figure 4.12: KKW Preprocessing Player. R is any finite commutative ring.

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

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

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

Figure 4.13: KKW Online Players. R is any finite commutative ring.

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

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

return s_0

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

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

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

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

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

Figure 4.14: Simulating the condensed views in KKW. R is the commutative ring over which the arithmetic circuits are computed.

Game AdaptiveDlog.

$\mathscr{A}$		Game
	$\xleftarrow{1 : h}$	$h \xleftarrow{\$} \mathbb{G}$
	$\xrightarrow{2 : g_*}$	
	$\xleftarrow{3 : g}$	$g \xleftarrow{\$} \mathbb{G}$
	$\xrightarrow{4 : x,y,z}$	
		Wins iff. $x,y,z \neq 0 \wedge h^x g_*^y g^z = 1$

Figure 4.15: AdaptiveDlog Game. Messages are label (in red) for easy referencing.

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

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

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

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

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

1 : $g_1 = \mathsf{ck}$
2 : **for** $i \in [2, \ell] : g_i \leftarrow \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$
3 : **return** $h^r g_1^{\mathbf{v}_1} g_2^{\mathbf{v}_2} \cdots g_\ell^{\mathbf{v}_\ell}$

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

1 : $g_1 = \mathsf{ck}$
2 : **for** $i \in [2, \ell] : g_i \leftarrow \mathsf{P}_{|\mathbb{G}|}(g_{i-1})$
3 : $r \leftarrow \mathsf{aux} - \sum_{i \in [\ell]} \mathsf{ek} \cdot (\mathbf{v}'_i - \mathbf{v}_i) \in \mathbb{Z}_{|\mathbb{G}|}$
4 : **return** r

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

1 : $E = [\ell] \setminus B = \{i_*\}$
2 : $\mathsf{ek} \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}; g_{i_*} \leftarrow h^{\mathsf{ek}}$
// Apply the inverse permutation $i_* - 1$ times to g_{i_*}.
3 : **for** $i \in [i_* - 1, 1] : g_i \leftarrow \mathsf{P}^{-1}_{|\mathbb{G}|}(g_{i+1})$
4 : $\mathsf{ck} = g_1$
5 : **return** $(\mathsf{ck}, \mathsf{ek})$

Figure 4.16: Optimized partially binding vector commitments from the hardness of discrete log in the random oracle model. Commitment keys ck and commitments com are both single group elements in $\mathbb{G}$. Openings consist of a single scalar $r \in \mathbb{Z}_{|\mathbb{G}|}$.

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

1 : Receive h from the AdaptiveDlog game.

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

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

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

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

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

A : Let $g_* = Q_q$

B : Send g_* to AdaptiveDlog; Receive g from AdaptiveDlog.

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

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

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

C : **return** R_q

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

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

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

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

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

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

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

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

9 : Define:

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

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

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

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

11 : Let:

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

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

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

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

Figure 4.17: Reduction for partially binding commitment scheme to discrete log in the programmable, invertible random oracle model.

Chapter 5

Curve Trees: Practical and Transparent Zero-Knowledge Accumulators

Matteo Campanelli, Mathias Hall-Andersen, Simon Holmgaard Kamp.

Orignally published at USENIX 2023.

Abstract

In this work we improve upon the state of the art for practical *zero-knowledge for set membership*, a building block at the core of several privacy-aware applications, such as anonymous payments, credentials and whitelists. This primitive allows a user to show knowledge of an element in a large set without leaking the specific element. One of the obstacles to its deployment is efficiency. Concretely efficient solutions exist, e.g., those deployed in Zcash Sapling, but they often work at the price of a strong trust assumption: an underlying setup that must be generated by a trusted third party.

To find alternative approaches we focus on a common building block: accumulators, a cryptographic data structure which compresses the underlying set. We propose novel, more efficient and fully transparent constructions (i.e., without a trusted setup) for accumulators supporting zero-knowledge proofs for set membership. Technically, we introduce new approaches inspired by "commit-and-prove" techniques to combine shallow Merkle trees and 2-cycles of elliptic curves into a highly practical construction. Our basic accumulator construction—dubbed *Curve Trees*—is completely transparent (does not require a trusted setup) and is based on simple and widely used assumptions (DLOG and Random Oracle Model). Ours is the first fully transparent construction that obtains concretely small proof/commitment sizes for large sets and a proving time one order of magnitude smaller than proofs over Merkle Trees with Pedersen hash. For a concrete instantiation targeting 128 bits of security we obtain: a commitment to a set of *any* size is 256 bits; for $|S| = 2^{40}$ a zero-knowledge

membership proof is 2.9KB, its proving takes 2s and its verification 40ms on an ordinary laptop.

Using our construction as a building block we can design a simple and concretely efficient anonymous cryptocurrency with full anonymity set, which we dub $\mathbb{V}$cash. Its transactions can be verified in $\approx$ 80ms or $\approx$ 5ms when batch-verifying multiple (> 100) transactions; transaction sizes are 4KB. Our timings are competitive with those of the approach in Zcash Sapling and trade slightly larger proofs (transactions in Zcash Sapling are 2.8KB) for a completely transparent setup.

5.1 Introduction

Zero-knowledge proofs are a cryptographic primitive that allows one to prove knowledge of a secret without revealing it. In many applications the focus is on proofs that are short and with efficient running time. One of the rising applications of zero-knowledge is in *set-membership*: given a short digest to a set S, we want to later show knowledge of a member in the set without revealing the latter. This primitive is useful in domains such as privacy-preserving distributed ledgers, anonymous broadcast, financial identities and asset governance (see, e.g., discussion in [BCF+21]).

Limitations of prior work. Our focus in this work is on solutions that are highly *practical*. That is, solutions with concretely short proving/verification time and short proofs. While efficient solutions to zero-knowledge set-membership already exist, we argue that they have limitations. In particular, either they still have a high computational/communication cost (we elaborate in Section 5.1 where we compare to transparent polynomial commitments and ring signatures [LRR+19]) or they rely on proof systems that are *non-transparent*. The latter means that, in order for the system to be bootstrapped, it is necessary to invoke a trusted authority. This is true for example in Zcash (Sapling) [HBHW21] and in [CFH+22]. While we can partly overcome this issue by emulating the trusted authority through a large-scale MPC, this is still highly expensive, both computationally and logistically[1]. Other solutions, such as [BCF+21, CHA21], mitigate this problem by requiring a trusted setup for parameters that are reusable in other cryptographic settings (an RSA modulus). This, however, still requires invoking a trusted authority or arranging a parameter-generation ceremony [CHI+20], which may not always be viable. We then turn to solutions that are fully transparent and still very efficient.

Our contributions.

Our main contribution is a concretely efficient construction for proving private set-membership with a fully transparent setup. Specifically we design a new data structure, CURVE TREES, that supports concretely small commitment to a set and where we can show set membership in zero-knowledge and with a small proof.

The design of a curve tree is simple and relies on discrete logarithm and the random oracle model (ROM) for its security. A curve tree can be described as a shallow Merkle

[1] `https://z.cash/technology/paramgen/`

tree where the leaves are points over an elliptic curve (and so are internal nodes). To hash, at each level we use an appropriately instantiated Pedersen hash alternating curves at each layer (we require a 2-cycle of curves). To prove membership in zero-knowledge we use commit-and-prove[2] capabilities of Bulletproofs and leverage the algebraic nature of our data structure. Our curves can be instantiated with existing ones in literature (see "Supported Curves" in Section 5.2). While we focus on accumulators and set membership, our approach can straightforwardly be applied to opening of vectors rather than sets obtaining an "index-hiding" vector commitment [ZBK+22].

For a concrete instantiation targeting 128 bits of security we obtain: a commitment to a set of *any* size is 256 bits; for $|S| = 2^{40}$ a zero-knowledge membership proof is 2.9KB, its proving takes 2s and its verification 40ms on an ordinary laptop.

Using our construction as a building block we can construct a simple and concretely efficient anonymous payment system with full anonymity set[3] and *transparent setup*. We dub this payment system $\mathbb{V}$Cash[4]. In $\mathbb{V}$Cash, the constraint system used for the zero-knowledge proof of a "spend" transaction is 20x smaller than that in Zcash Sapling.

The main distinguishing feature of $\mathbb{V}$Cash is that it can be concretely efficient and still support *full anonymity sets*. The latter is roughly the subset of existing transactions a spent transaction can be narrowed down to (if a protocol supports a full anonymity set then this set consists of the whole history of transactions so far). For "two inputs/two outputs" settings and for anonymity sets of size 2^{32} (like in Zcash) our confidential transactions ($\mathbb{V}$cash) require participants to compute/verify two Bulletproofs proofs of < 5000 constraints each. Verifying each of the proofs in parallel (4 cores) in batches of at least 100 transactions (e.g. when verifying the validity of all transactions in a block) yields a very practical per-transaction verification time of ≈ 5 ms. Transaction sizes are 4 KB. Our timings are competitive with those of the approach in Zcash Sapling and trade slightly larger proofs for a completely transparent setup and simpler curve requirements.

As a side contribution, we provide the first optimized implementation of Bulletproofs that can be instantiated with arbitrary curves and supports vector commitments of arbitrary dimension and arbitrary computations at the same time. To the best of our knowledge, previous implementations were not written modularly to work with arbitrary curves or supported only specific computations, such as range proofs.

STRUCTURE-PRESERVING FEATURES. From the theoretical side, one interesting feature of curve trees is their *structure-preserving* properties[ACD+16]. This means our construction never needs to use any combinatorial hash (e.g., SHA2) or use their bit decomposition, but it only relies on basic structural properties of groups. In

[2] In the sense of the commit-and-prove building blocks in LegoSNARK [CFQ19] and in the work by Lipmaa [Lip16].

[3] An anonymity set can be seen as the subset of existing transactions a spent transaction can be narrowed down to. We say that a protocol supports a *full* anonymity set if the set consists of the whole history of transactions.

[4] As a reference to both Zcash and $\mathbb{V}$eksel [CHA21] from which it borrows part of its design.

this sense, this construction provides some nuances to the implications of the recent impossibility result in [CFGG22].

Related Work

Zero-Knowledge Sets.

Seminal work by Micali, Rabin and Kilian [MRK03] introduce the notion of a "zero-knowledge set": a hiding commitment to a set, enabling membership and non-membership proofs. Note that this is exactly complementary to the goal of this paper: in zero-knowledge sets, the *set is hidden* and the *retrieved elements public*, here the *set is public* and the *retrieved element hidden*. In Camenisch-Stadler notation (Section 5.2) this relation is $\{(S,r) : c = \mathsf{Com}(S;r) \wedge x \in S\}$ instead of $\{(x,r) : c = \mathsf{Com}(x;r) \wedge x \in S\}$.

Highly efficient constructions of zero-knowledge sets are known under a range of assumptions, notably Chase et al. [CHL$^+$05] generalize the original construction by Micali et al. using Mercurial commitments.

Accumulators from Groups of Unknown Order.

The original work by Benaloh and de Mare [Bd94] introducing cryptographic accumulators provides a simple construction based on strong RSA: a set of prime integers are accumulated by iteratively exponentiation in an RSA group. Camenisch and Lysyanskaya [CL02] extended this accumulator to be dynamic, Baldimtsi et al. [BCD$^+$17] generically obtaining an adaptively sound dynamic accumulator by combining 1) an adaptively sound positive additive accumulator and 2) a non-adaptively sound positive dynamic accumulator. Rather than RSA, these constructions can be can instantiated with class groups: which avoids the need for a trusted setup, ut incurs a sustantial $\approx 20\times$ computational overhead at the same security level.

Accumulators from Bilinear Pairings.

Nguyen constructed accumulator from bilinear pairings [Ngu05], this construction was subsequently extended by Damgård and Triandopoulos [DT08] to support non-membership proofs. More recently Ghosh et al. [GOP$^+$16] showed how to prove membership in zero-knowledge. In the concurrent work Zapico et al. [ZBK$^+$22] reduces the computational cost of proving membership from $O(n)$ to $O(\log(n))$, by relying on an $O(n \log n)$ precomputation. Common for all these works is the reliance on a structured "powers-of-τ" style structured reference string (SRS): size of the public parameters is proportional to the (apriori bounded) maximum set size and knowledge of the trapdoor breaks binding.

Authenticated Hash Tables & Verkle Trees.

Charalapos, et al. [PTT08] suggests using a tree of accumulators: where every internal node is a cryptographic accumulator containing all its children. Which allows a trade-

off between membership proof size and cost of updating the accumulator. The same concept ("Verkle Trees") was subsequently independently rediscovered by industry-afflicated people [BFB21]. We observed that Charalapos et al. holds a patent for this construction [CP14] when used for authenticated computation, it is unclear if this applies to the scheme as used in the Ethereum blockchain. Our work differs from these by not using an accumulator at each level, the compression function is a simple Pedersen commitment, furthermore these works do not allow/describe efficient zero-knowledge membership proofs.

Curve Trees and Algebraic Merkle Trees.

The closest related (zero-knowledge) accumulator is the approach taken in Zcash (Sapling and Orchard versions) [HBHW21], in which a Merkle tree is instantiated with a hash function admitting an efficient algebraic description. In case of Zcash this hash function is based on multi-scalar exponentation over specially chosen elliptic curves. For "Pedersen hashes" as used in Zcash Sapling, the resulting circuit requires ≈ 44000 constraints (multiplications) for memberships of size 2^{32}, Our approach, on the other hand, requires proving ≈ 4500 constraints in zero-knowledge; roughly an order of magnitude less. Merkle trees instantiated with "SNARK-friendly hash functions" (e.g. Poseidon [GKR+21]) has similar performance compared to ours (see Section 5.8), however the concrete security of these hash functions is less well understood [BGL20] [rep20].

Halo2 and Recursive Proofs.

Halo2[5] is a transparent (zero-knowledge) proof system enabling efficient recursion using "atomic accumulation" and cycles of elliptic curves. For efficiency the curves used by Halo2 need to have a "smooth" multiplicative subgroup to perform FFT which rules out some curves, in particular the `secp256k1` / `secq256k1` cycle (instead supported by our Bulletproofs implementation). This requirement restricts Halo2's compatibility with systems using other curves.

Although both—curve trees and Halo2—rely on the special algebraic structure of a cycle of curves, their goals are orthogonal: Halo2 is a proof system, ours a specialized data structure for zero-knowledge for set membership. Our techniques rely on a commit-and-prove which we instantiate with Bulletproofs for easy comparison.[6] It is possible to instantiate our scheme with Halo2; Halo 2 is ultimately *not* a competing approach but a potential way to apply the Curve Tree framework. However Halo2's generalized PLONK-based arithmetization [GWC19] enables a more complex set of potential optimizations, including custom gates and lookups, which makes an apple-to-apple comparison substantially harder. We believe that replacing Bulletproofs with Halo2 would improve concrete performance: via custom gates for ECC operations and tables of precomputed points.

[5] `https://electriccoin.co/blog/explaining-halo-2/`

[6] The Bulletproof arithmetization is R1CS, hence comparing the number of constraints is easy

5.2 Preliminaries

Familiarity with elliptic curves and non-interactive proof systems is a prerequisite for this paper and in this section we provide a brief (and incomplete) introduction to these subjects. Since our techniques will only apply to elliptic curves we do not generalize to other group structures.

Elliptic Curves

We denote by $\mathbb{E}[\mathbb{F}_q] \subseteq \mathbb{F}_q \times \mathbb{F}_q$ the set of points in $(\mathtt{x}, \mathtt{y})$ on the elliptic curve $\mathbb{E}$ [Mil86]. We denote points on elliptic curves using upper-case letters (e.g. G and H). Whenever clear from context we might omit the *base field* $\mathbb{F}_q$ and simply write $\mathbb{E}$. The curve points form an Abelian group $(\mathbb{E}[\mathbb{F}_q], +)$; we use "additive notation". Throughout this paper, the number of points on $\mathbb{E}[\mathbb{F}_q]$ denoted $p := |\mathbb{E}[\mathbb{F}_q]|$ will always be prime, hence the group is cyclic. We call the prime field $\mathbb{F}_p \cong \mathbb{Z}/(p\mathbb{Z})$ the *scalar field* of $\mathbb{E}[\mathbb{F}_q]$ and denote by $[s] \cdot G$, with $s \in \mathbb{F}_p$ and $G \in \mathbb{E}[\mathbb{F}_q]$, s acting on G in the $\mathbb{Z}$-module ("scalar multiplication"). We denote by $\langle \mathbf{s}, \mathbf{G} \rangle = \sum_i [s_i] \cdot G_i$ the "inner product" between a vector of scalars $\mathbf{s} \in \mathbb{F}_p^n$ and a list of group elements $\mathbf{G} \in \mathbb{E}[\mathbb{F}_q]^n$.

Assumption: Generalized Discrete-Log

We rely on a common variant of the discrete logarithm assumption for multiple generators over elliptic curves:

Assumption 2 (Generalized Discrete-Log) *Let $\mathscr{G}(1^\lambda)$ a procedure for sampling a new elliptic curve. For all PPT adversaries $\mathscr{A}$ and $m \geq 2$:*

$$\Pr\left[\begin{array}{l} \langle \mathbf{a}, \mathbf{G} \rangle = 0 \in \mathbb{E}[\mathbb{F}_q] \\ \quad \wedge\, \mathbf{a} \neq \mathbf{0} \in (\mathbb{F}_p)^m \end{array} : \begin{array}{r} (\mathbb{E}, \mathbb{F}_q, \mathbb{F}_p) \leftarrow \mathscr{G}(1^\lambda) \\ \mathbf{G} \xleftarrow{\$} \mathbb{E}[\mathbb{F}_q]^m \\ \mathbf{a} \leftarrow \mathscr{A}((\mathbb{E}, \mathbb{F}_q, \mathbb{F}_p), \mathbf{G}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

We refer to this assumption as DLOG throughout the paper. Note that generalized variant of DLOG has a tight reduction to the standard ($m = 2$) variant.

Pedersen Commitments

Throughout the paper we will rely on the ubiquitous Pedersen commitment scheme. The setup consists of $(\mathbb{E}, \mathbb{F}_p, \mathbb{F}_q, \ell, \mathbf{G}, H)$, with $\mathbb{F}_p = |\mathbb{E}[\mathbb{F}_q]|$, $G_1, \ldots, G_\ell, H \in \mathbb{E}[\mathbb{F}_q]$. The commitment to $\mathbf{v} \in \mathbb{F}_p^\ell$ with randomness r is computed as follows:

$$C = \mathsf{Com}(\mathbf{v}; r) = \langle \mathbf{v}, \mathbf{G} \rangle + [r] \cdot H \in \mathbb{E}[\mathbb{F}_q]$$

It is easy to see that computational binding follows from DLOG (Assumption 2). Hiding is perfect and follows from the observation that $[r] \cdot H$ with $r \xleftarrow{\$} \mathbb{F}_p$ is uniformly

distributed over the group $\mathbb{E}[\mathbb{F}_q]$. Importantly, Pedersen commitments are *rerandomizable commitments*: sampling $\delta \xleftarrow{\$} \mathbb{F}_p$ and computing $C^* \leftarrow C + [\delta] \cdot H$ yields a commitment to the same $\mathbf{v}$ with randomness $r + \delta$, furthermore the distribution of C^* is independent of C: it is a "fresh" perfectly hiding commitment to the same value.

Avoiding Bit Decomposition via 2-Cycles of Curves

A 2-cycle of elliptic curves consists of two elliptic curves $\{\mathbb{E}_{(\text{evn})}, \mathbb{E}_{(\text{odd})}\}$ and two prime fields $\{\mathbb{F}_p, \mathbb{F}_q\}$ such that:

$$p = |\mathbb{E}_{(\text{evn})}[\mathbb{F}_q]| \text{ And } q = |\mathbb{E}_{(\text{odd})}[\mathbb{F}_p]|$$

In other words: the base/scalar fields of the two curves are complementary. Crucial for our application will be the observation that a point $(\mathbb{x}, \mathbb{y}) \in \mathbb{E}_{(\text{evn})}[\mathbb{F}_q]$ can be treated as a pair of scalars on $\mathbb{E}_{(\text{odd})}$, e.g. $[\mathbb{x}] \cdot G_1 + [\mathbb{y}] \cdot G_2 \in \mathbb{E}_{(\text{odd})}[\mathbb{F}_p]$ for $G_1, G_2 \in \mathbb{E}_{(\text{odd})}[\mathbb{F}_p]$ is a well-defined operation. The observant reader will see that this defines Pedersen commitments in $\mathbb{E}_{(\text{odd})}$ to lists of points on $\mathbb{E}_{(\text{evn})}$, without relying on bit-decomposition for field elements or hashing, making it cheaper in zero-knowledge. Numerous instantiations of 2-cycles exists, e.g., the Pasta cycle [Hop20] (used in this paper and Halo2) or the well known `secp256k1` / `secq256k1` cycle [7]. No known attacks make use of this additional structure, additionally we do not require any efficiently computable pairings on either curve.

Non-Interactive Zero-Knowledge Proofs

Camenisch-Stadler Notation

When expressing an NP relation $R(x, w)$ we use a variant of Camenisch-Stadler notation[CS97], the witness w is explicitly (enclosed in brackets) and the public statement x is defined by all remaining terms e.g. the "discrete log relation" $\mathscr{R} \coloneqq \{(z) : y = [z] \cdot G\}$ – the witness is the scalar $z \in \mathbb{F}_p$, while group elements $G, y \in \mathbb{E}$ constitute the instance.

Non-Interactive Zero-Knowledge Arguments-of-Knowledge (NIZKAoKs)

Definition 17 *A NIZKAoK for a relation family $\mathfrak{R} = \{\mathfrak{R}_\lambda\}_{\lambda \in \mathbb{N}}$ is a tuple of algorithms $\mathsf{ZK} = (\mathsf{Prove}, \mathsf{VerProof})$ with the following syntax:*

- $\mathsf{ZK.Prove}(\mathsf{urs}, R, x, w) \to \pi$ *takes as input a string* urs*, a relation description R, a statement x and a witness w such that $R(x, w)$; it returns a proof π.*
- $\mathsf{ZK.VerProof}(\mathsf{urs}, R, x, \pi) \to b \in \{0, 1\}$ *takes as input a string* urs*, a relation description R, a statement x and a proof π; it accepts or rejects the proof.*

[7] With `secp256k1` being used by the Bitcoin blockchain.

Non-Interactive Zero-Knowledge schemes (or NIZKs) require a reference string which can be either uniformly sampled (a urs), or structured (a srs). In the latter case it needs to be sampled by a trusted party. In this work we use and assume transparent NIZKAoKs, i.e. whose algorithms use a reference string urs *sampled uniformly.*

We require a NIZKAoK to be complete, that is, for any $\lambda \in \mathbb{N}, R \in \mathfrak{R}$ *and* $(x,w) \in R$ *it holds with overwhelming probability that* $\mathsf{VerProof}(\mathsf{urs}, R, x, \pi)$ *where* $\mathsf{urs} \xleftarrow{\$} \{0,1\}^{\mathsf{poly}(\lambda)}$ *and proof* $\pi \leftarrow \mathsf{Prove}(\mathsf{urs}, R, x, w)$*. For security we require standard notions of knowledge-soundness and zero-knowledge:*

Knowledge-Soundness. *For all* $\lambda \in \mathbb{N}$ *and for all (non-uniform) efficient adversaries* $\mathscr{A}$*, there exists a (non-uniform) efficient extractor* $\mathscr{E}$ *such that*

$$\Pr\left[\begin{array}{l} \mathsf{urs} \xleftarrow{\$} \{0,1\}^{\mathsf{poly}(\lambda)}; \\ (x,\pi) \leftarrow \mathscr{A}(\mathsf{urs}) \\ w \leftarrow \mathscr{E}(\mathsf{urs}) \end{array} : \begin{array}{r} R_\lambda(x,w) \neq 1 \wedge \\ \mathsf{Vfy}(\mathsf{urs}, x, \pi) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Note the order of quantifiers: the extractor $\mathscr{E}$ *depends on* $\mathscr{A}$*.*

Zero-Knowledge. *There exists a PPT simulator* $\mathscr{S}$ *such that for any* $\lambda \in \mathbb{N}$*, PPT* $\mathscr{A}$*, relation* $R \in \mathfrak{R}$*,* $(x,w) \in R$*, it holds* $p_0 = p_1$ *where:*

$$p_b := \Pr\left[\begin{array}{l} \mathsf{urs}_1 \xleftarrow{\$} \{0,1\}^{\mathsf{poly}(\lambda)} \\ (\mathsf{urs}_0, \pi_0) \leftarrow \mathscr{S}(1^\lambda, x) \\ \pi_1 \leftarrow \mathsf{Prove}(\mathsf{urs}, x, w) \end{array} : \mathscr{A}(1^\lambda, \mathsf{urs}_b, \pi_b) = 1\right]$$

Remark 6 (Practical Efficiency) *For a broad class of NIZKs the "cost" of the NZIK*[8] *scales with the number of multiplicative constraints in the relation. Hence when comparing/estimating how "expensive" a certain relation is prove using a NIZK, the number of multiplications is a broadly useful metric which translates to concrete performance for a wide range of NIZKs.*

Commit-and-Prove for Pedersen Commitments

The techniques in this paper rely heavily on efficient "commit-and-prove" NIZKs for Pedersen commitments (Section 5.2). A "commit-and-prove" (C&P) NIZKs for Pedersen commitments enable efficient proofs of relations in which (part of) the witness is additionally committed inside a pedersen commitment, i.e. relations of the form:

$$R^* := \{(\mathbf{w}, r) : C = \mathsf{Com}(\mathbf{w}; r) \in \mathbb{E}[\mathbb{F}_q] \wedge R(x, \mathbf{w}) = 1\}$$

Many efficient "commit-and-prove" NIZKs exists e.g. Bulletproofs[BBB+18], Compressed Σ-Protocols[AC20] and Halo2. All these schemes make black-box use of the group $\mathbb{E}[\mathbb{F}_q]$, i.e. avoid expressing the group operation as an NP relation over $\mathbb{F}_q$. Note that in the example above $\mathbf{w} \in \mathbb{F}_p^\ell$, where $\mathbb{F}_p$ is the scalar field of $\mathbb{E}[\mathbb{F}_q]$.

[8] In prover time, verifier time or proof size, depending on the concrete NIZK.

A Concrete C&P-NIZKAoK: Bulletproofs

We denote by zk-BP$[\mathbb{E}]$ an instantiation of the Bulletproofs [BBB+18] NIZKAoK on the elliptic curve $\mathbb{E}$. The Bulletproofs scheme exbibits the following relevant properties: (1) It is a commit-and-prove for Pedersen commitments on $\mathbb{E}[\mathbb{F}_q]$ and an concretely efficient proof system for R1CS relations over $\mathbb{F}_p$. (2) The URS consists of a list of random group elements in $\mathbb{E}$ with size linear with the size of the relation being proved. (3) zk-BP$[\mathbb{E}]$ is computationally (simulation) sound in the random oracle model under the DLOG assumptions (Assumption 2) on $\mathbb{E}[\mathbb{F}_q]$.

5.3 Zero-Knowledge Set Membership

In this section we describe a modular primitive for proving set memberships in zero-knowledge, which can be composed with commit-and-proof zero-knowledge proof system to prove additional properties about the member of the set. Informally, for a set of rerandomizable commitments (see Section 5.2) $S = \{C_1, \ldots, C_n\}$ the primitive proves:

$$\{(r,i) : \hat{C} = \mathsf{Rerand}(C_i, r)\}$$

In other words, the commitment $\hat{C}$ is a rerandomization of *a commitment* in S, without revealing which. Additional properties about the opening of $\hat{C}$ can then be proved using commit-and-prove techniques (see Section 5.2).

Need for Compression. The relation outlined above has size n, as a result verifying a proof for the relation requires $O(n)$ work – to even read the statement. To reduce this cost, the set of commitments itself can be compressed using a commitment. When the set is fixed, or incrementally updated, this greatly reduces computation for both prover and verifier. We formalize this general primitive below. Our scheme achieves $O(\log(n))$ communication and $O(\sqrt[\mathsf{D}]{n})$ computation where D is a parameter of the scheme (D is both constant and small) and n is the size of the set. [9]

Select-and-Rerandomize Accumulators

Below, we fix the message space of the commitment scheme to $\mathbb{F}^k$ for some k and its randomness space to $\mathbb{F}$ – as is the case for Pedersen commitments (Section 5.2). Our definitions below can be generalized easily.

Definition 18 (Select-and-Rerandomize) *A select-and-rerandomize accumulator scheme consists of six algorithms:*

$\mathsf{SelRerand.Setup}(1^\lambda) \to \mathsf{pp}$ *returns public parameters of the scheme. These parameters are transparent—no trusted party needs be invoked.*

$\mathsf{SelRerand.Comm}(\mathsf{pp}, v_{\text{leaf}}, o) \to C$ *commits to a string v_{leaf} with randomness o.*

[9] The circuit has $O(\sqrt[\mathsf{D}]{n})$ constraints for each layer, but as D is constant this does not affect the asymptotic complexity. Similarly the size of the proof is $\Theta(\log(\sqrt[\mathsf{D}]{n})) = \Theta(\log n)$

$\mathsf{SelRerand.Rerand}(\mathsf{pp}, C, r) \to \hat{C}$ *rerandomizes committment C with randomness r.*

$\mathsf{SelRerand.Accum}(\mathsf{pp}, S) \to A$ *deterministically accumulates a set of commitments. We assume the set S to have a canonical order.*

$\mathsf{SelRerand.Prove}(\mathsf{pp}, S, C, r) \to \pi$ *returns a proof showing that $C \in S$ verifiable through a rerandomized commitment to c with randomness r.*

$\mathsf{SelRerand.Vfy}(\mathsf{pp}, A, \hat{C}, \pi) \to 0/1$ *verifies that $\hat{C}$ is a rerandomization of an element in the set.*

Correctness of Select-and-Rerandomize. For any $\lambda \in \mathbb{N}$, for any set $S = \{v_i\}_i$, $j^* \in [|S|]$, commitment randomness $(o_1, \ldots, o_n)$ and commitment rerandomization r the verification always succeeds, i.e.

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{SelRerand.Setup}(1^\lambda) \\ c_i \leftarrow \mathsf{SelRerand.Commit}(\mathsf{pp}, v_i, o_i)\ i = 1, \ldots, n \\ A = \mathsf{SelRerand.Accum}(\mathsf{pp}, \{C_1, \ldots, C_n\}) \ : \ \mathsf{SelRerand}.\mathscr{V}(\mathsf{pp}, A, \hat{C}, \pi) = 1 \\ \hat{C} = \mathsf{SelRerand.Rerand}(\mathsf{ck}, C_{j^*}, r) \\ \pi \leftarrow \mathsf{SelRerand.Prove}(\mathsf{pp}, S, C_{j^*}, r) \end{array}\right] = 1$$

The above can be thought as a main correctness property. For it to be meaningful, it needs to be complemented by the following one, which specifically makes explicit what it means for commitments (output of Comm) to be rerandomizable: for any $\lambda \in \mathbb{N}$, for any message $m \in \mathbb{F}^k$, opening o and randomness $r \in \mathbb{F}$, it should hold that $\mathsf{SelRerand.Rerand}(\mathsf{pp}, \mathsf{Comm}(\mathsf{pp}, m; o)) = \mathsf{Comm}(\mathsf{pp}, m; o + r)$ and $\mathsf{pp} \leftarrow \mathsf{SelRerand.Setup}(1^\lambda)$[10].

For our application/instantiation we require the select-and-rerandomize scheme to satisfy the following security notions:

Select-and-Rerandomize Binding. This is the main security definition of our model. We say the select-and-rerandomize scheme is binding if there exists a negligible function $\mathsf{negl}(\lambda)$ such that for any PPT adversary $\mathscr{A}$:

Perfect Hiding of Commitment. For all m, m', $\mathsf{pp} \leftarrow \mathsf{SelRerand.Setup}(1^\lambda)$ the following distributions are perfectly indistinguishable:

$$\{\mathsf{SelRerand.Comm}(\mathsf{pp}, m, o) \mid o \xleftarrow{\$} \mathbb{F}\}$$
$$\approx \{\mathsf{SelRerand.Comm}(\mathsf{pp}, m', o') \mid o' \xleftarrow{\$} \mathbb{F}\}$$

Select-and-Rerandomize Zero-Knowledge. A select-and-rerandomize is (perfect) zero-knowledge if there exists an efficient simulator $\mathscr{S}$, such that for any $\lambda \in \mathbb{N}$, any

[10] Notice that homomorphic commitments (and thus Pedersen commitments) satisfy this property.

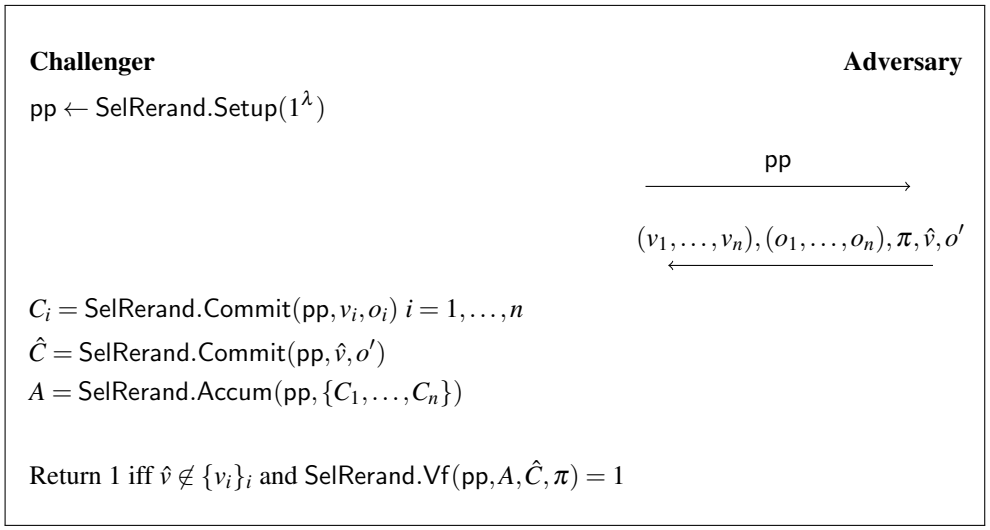

Figure 5.1: Security experiment for zero-knowledge set membership

(stateful) adversary $\mathscr{A}$, any $j^* \in [n]$, it holds $p_0 = p_1$ where

$$p_b := \Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{SelRerand.Setup}(1^\lambda); \\ (v_1,\ldots,v_n,o_1,\ldots,o_n) \leftarrow \mathscr{A}(\mathsf{pp}) \\ S := \{C_i = \mathsf{SelRerand.Commit}(\mathsf{pp},v_i,o_i)\}_{i\in[n]} \\ r \xleftarrow{\$} \mathbb{F}; \hat{C} = \mathsf{SelRerand.Rerand}(\mathsf{pp},C_{j^*},r) \\ \pi \leftarrow X_b(\mathsf{pp},S,C,\hat{C},r) \\ \mathscr{A}(\mathsf{pp},\hat{C},\pi) = 1 \end{array}\right]$$

With $X_0(\mathsf{pp},S,C,\hat{C},r) := \mathscr{S}(\mathsf{pp},S,\hat{C})$ and $X_1(\mathsf{pp},S,C,\hat{C},r) := \mathsf{SelRerand.Prove}(\mathsf{pp},S,C,r)$.

Remark 7 *Our formalization of* SelRerand *combines together commitments, accumulators (Definition 23) and zero knowledge properties. There are, of course, other possible way to model this primitive. We found this natural enough. We also observe that our definition of correctness and binding imply their counterparts in a standard accumulator as a special case (where the commitment and the rerandomization are trivial).*

5.4 Curve Trees as Accumulators

In this section we first define curve trees. We then describe some of its properties in terms of commitments (that are binding and hiding). Finally, we show how to traverse a tree to show membership of a an element in zero-knowledge. The latter represents our actual construction (Figure 5.2).

Intro to $(\ell, \mathbb{E}_{(\text{evn})}, \mathbb{E}_{(\text{odd})})$-Curve Trees

Recall (from Section 5.2) the observation that $[\mathbb{x}] \cdot G_1 + [\mathbb{y}] \cdot G_2 \in \mathbb{E}_{(\text{odd})}$ for any $(\mathbb{x}, \mathbb{y}) \in \mathbb{E}_{(\text{evn})}$[11] is a meaningful operation. This is generalizable to any number ℓ of $\mathbb{E}_{(\text{evn})}$ points: computing $\langle \mathbb{x}, \mathbf{G}^{\mathbb{x}}_{\mathbb{E}_{(\text{odd})}} \rangle + \langle \mathbb{y}, \mathbf{G}^{\mathbb{x}}_{\mathbb{E}_{(\text{odd})}} \rangle \in \mathbb{E}_{(\text{odd})}$ for $i \in [\ell] : (\mathbb{x}_i, \mathbb{y}_i) \in \mathbb{E}_{(\text{evn})}$. This is a compression function $f_{\mathbb{E}_{(\text{odd})}} : {\mathbb{E}_{(\text{evn})}}^{\ell} \mapsto \mathbb{E}_{(\text{odd})}$. At this point a natural strategy to obtain an accumulator is to use $f_{\mathbb{E}_{(\text{odd})}}$ to construct a Merkle tree from $f_{\mathbb{E}_{(\text{evn})}}$: a tree in which every parent (an $\mathbb{E}_{(\text{odd})}$ point) is the hash of its children ($\mathbb{E}_{(\text{evn})}$ points) using $f_{\mathbb{E}_{(\text{odd})}}$. However this encounters an obvious "type problem": the output of $f_{\mathbb{E}_{(\text{odd})}}$ is a point on $\mathbb{E}_{(\text{odd})}$, while the inputs are points on $\mathbb{E}_{(\text{evn})}$, preventing us from applying $f_{\mathbb{E}_{(\text{odd})}}$ to the resulting outputs. The solution to this "type mismatch" is to define $f_{\mathbb{E}_{(\text{evn})}} : {\mathbb{E}_{(\text{odd})}}^{\ell} \mapsto \mathbb{E}_{(\text{evn})}$ analogously to $f_{\mathbb{E}_{(\text{odd})}}$ and *alternate the compression function* at every level of the tree. We call this construction a *Curve Tree*, which can be seen as an "algebraically compatible" Merkle tree using Pedersen commitments alternating over $\mathbb{E}_{(\text{evn})}/\mathbb{E}_{(\text{odd})}$: a parent node on one curve will be the hash of its children, represented as points on the other curve. To refer more easily to curves alternating within a tree, we introduce the following piece of notation.

Remark 8 (Notation for alternating curves) *As mentioned above, a curve tree alternates between curves at each level. If we are referring to a specific "current" level (obvious from context), we will denote the corresponding curve as $\mathbb{E}_{(\_)}$. The "other" curve will be denoted by $\mathbb{E}_{\text{other}(\_)}$. That is: if $\mathbb{E}_{(\_)}$ is $\mathbb{E}_{(evn)}$, then $\mathbb{E}_{\text{other}(\_)}$ is $\mathbb{E}_{(odd)}$, and vice versa. We extend this notation to subscripts for group elements in the natural way (see, e.g., usage in the following definition).*

In order to define a Curve Treewe adopt a variant of (standard) approaches to defining a tree as a recursive data structure: an internal node is a list of (a function of) its children. The function which maps children to parents that we adopt uses an intermediate "labeling" step. A label can be thought of as a group element succinctly describing the node.

Definition 19 (Curve Trees) *A Curve Tree is parameterized by (I). a depth $\mathsf{D} \in \mathbb{N}$, (II). a branching factor $\ell \in \mathbb{N}$, (III). a 2-cycle of Elliptic curves $(\mathbb{E}_{(evn)}, \mathbb{E}_{(odd)}, \mathbb{F}_p, \mathbb{F}_q)$ (IV). 2ℓ points $\mathbf{G}^{\mathbb{x}}_{(evn)}, \mathbf{G}^{\mathbb{y}}_{(evn)} \in \mathbb{E}^{\ell}_{(evn)}$ (V). 2ℓ points $\mathbf{G}^{\mathbb{x}}_{(odd)}, \mathbf{G}^{\mathbb{y}}_{(odd)} \in \mathbb{E}^{\ell}_{(odd)}$.*

The tree is defined recursively over D a follows:

Leaves: $(0, \ell, \mathbb{E}_{(\_)}, \mathbb{E}_{\text{other}(\_)})-\mathsf{CurveTree}$*:*

A leaf node is completely described by a curve point $C \in \mathbb{E}_{(\_)}$. The label of a leaf is C.

Parents: $(\mathsf{D}, \ell, \mathbb{E}_{(\_)}, \mathbb{E}_{\text{other}(\_)})-\mathsf{CurveTree}$*:*

An internal node C is a list of ℓ $(\mathsf{D}-1, \ell, \mathbb{E}_{\text{other}(\_)}, \mathbb{E}_{(\_)})$-Curve Trees. Let $C_1 =$

[11] Assuming the identity ("point-at-infinity") is represented in $\mathbb{F}_q \times \mathbb{F}_q$

$(\mathbb{x}_1, \mathbb{y}_1) \in \mathbb{E}_{\mathsf{other}(\_)}, \ldots, C_\ell = (\mathbb{x}_\ell, \mathbb{y}_\ell) \in \mathbb{E}_{\mathsf{other}(\_)}$ *be their respective labels. The label* $C \in \mathbb{E}_{(\_)}$ *for the internal node is then defined as:*

$$C = \langle \langle \mathbb{x} \rangle, \mathbf{G}^{\mathsf{x}}_{(\_)} \rangle + \langle \langle \mathbb{y} \rangle, \mathbf{G}^{\mathsf{y}}_{(\_)} \rangle \tag{5.1}$$

(Note that the curves are switched between levels)

Trees for Sets. When we say that a curve tree is built for a set $S \subseteq \mathbb{E}$ (of size ℓ^{D}) we mean the natural layer-by-layer algorithm inductively constructing a tree with S as the leafs: partitioning S into subsets of size ℓ in some fixed way, then computing a Curve Tree for each set in the partition and forming a parent for the resulting ℓ children.

Binding and Hiding within Curve Trees

The previous notion (Definition 19) uses 2ℓ points per curve ($\mathbf{G}^{\mathsf{x}}_{(\mathsf{evn})}, \mathbf{G}^{\mathsf{y}}_{(\mathsf{evn})} \in \mathbb{E}^{\ell}_{(\mathsf{evn})}$ and $\mathbf{G}^{\mathsf{x}}_{(\mathsf{odd})}, \mathbf{G}^{\mathsf{y}}_{(\mathsf{odd})} \in \mathbb{E}^{\ell}_{(\mathsf{odd})}$) in order to label parent nodes by compressing their children. This already achieves a form of binding. By sampling one additional point per curve– $H_{(\mathsf{odd})} \in \mathbb{E}_{(\mathsf{odd})}, H_{(\mathsf{evn})} \in \mathbb{E}_{(\mathsf{evn})}$– we can blind / rerandomize a Curve Tree in the natural way. The root of a tree (and of each subtree) thus becomes a Pedersen commitment that is both binding and hiding. We formalize these observations below:

Lemma 8 *Assuming DLOG (Assumption 2) on* $\mathbb{E}_{(evn)}$ *and* $\mathbb{E}_{(odd)}$*, the root* $C \in \mathbb{E}_{(\_)}$ *of a* $(\mathsf{D}, \ell, \mathbb{E}_{(\_)}, \mathbb{E}_{\mathsf{other}(\_)})$*–CurveTree is a (non-hiding) Pedersen commitment whose opening is the* ℓ *roots of its children (in* $\mathbb{E}_{\mathsf{other}(\_)}$*). Additionally, for the same* C *and a random scalar* r*, the group element* $\hat{C} := C + [r] \cdot H_{(\_)}$ *is a hiding Pedersen commitment to* C*'s children.*

Proof 13 *The first part is a direct implication of the definition above. Also, observe then any internal node is already a root to a subtree. Let* r' *be a scalar (in the appropriate field) and let* $\hat{C} = C + [r'] \cdot H_{(\_)}$*. From standard properties of Pedersen commitments, we can observe* $\hat{C}$ *is still bound to the children of* C*. Hiding follows immediately (see Section 5.2).*

Traversing $(\ell, \mathbb{E}_{(\mathsf{evn})}, \mathbb{E}_{(\mathsf{odd})})$-Curve Trees

We now extend upon the observation in Section 5.4 that a node in a tree can be rerandomized. A natural strategy which stems from this observations is that, to prove membership of a Curve Tree in zero-knowledge, we can descend the tree one layer at a time starting from the root and following this approach: open a (hiding) commitment to a $(\mathsf{D}, \ell, \mathbb{E}_{(\_)}, \mathbb{E}_{\mathsf{other}(\_)})$-Curve Tree, pick one of it children (in zero-knowledge), then rerandomize the child and "output" the resulting hiding commitment to the $(\mathsf{D}-1, \ell, \mathbb{E}_{\mathsf{other}(\_)}, \mathbb{E}_{(\_)})$-Curve Tree; apply recursion. In this section we formalize a more efficient version of this informal sketch.

Descending a Single Level of the Tree

Our central component is a simple construction for a select-and-rerandomize-like relation for a *single* level in a curve tree. We later apply this at many levels at once in order to obtain a full select-and-rerandomize (Figure 5.2). Consider a curve tree whose internal nodes at layer $d-1$ are in $\mathbb{E}_{(\_)}$. The inputs to relation $\mathscr{R}^{(\mathsf{single\text{-}level},(\mathrm{evn}))}$ (resp. $\mathscr{R}^{(\mathsf{single\text{-}level},(\mathrm{odd}))}$) are:

- public inputs: a rerandomized commitment $\hat{C} \in \mathbb{E}_{(\mathrm{odd})}$ (resp. $\mathbb{E}_{(\mathrm{evn})}$); its alleged parent $C \in \mathbb{E}_{(\mathrm{evn})}$ (resp. $\mathbb{E}_{(\mathrm{odd})}$);
- witnesses: index i whose semantics is "$\hat{C}$ is the (rerandomized) i-th child of C"; Pedersen opening scalars $r, \delta, \mathtt{x}, \mathtt{y}$.

At each layer, this relation opens the parent commitment C to $\langle \mathtt{x} \rangle, \langle \mathtt{y} \rangle$ using a commit-and-prove over $\mathbb{E}_{(\_)}$, plus it shows rerandomization of one of the children. At each level, even or odd, it is defined as follows.

Definition 20 (Relation for Select-and-Rerandomize) *Define the following NP relation:*

$$\mathscr{R}^{(\mathsf{single\text{-}level},\mathbb{E}_{(\_)})} := \left\{ \begin{pmatrix} i, r, \delta, \\ \mathtt{x}, \mathtt{y} \end{pmatrix} : \begin{array}{c} \textit{// open parent} \\ C = \langle [\langle \mathtt{x} \rangle], \mathbf{G}^{\mathtt{x}}_{(\_)} \rangle \\ + \langle [\langle \mathtt{y} \rangle], \mathbf{G}^{\mathtt{y}}_{(\_)} \rangle \\ + [r] \cdot H_{(\_)} \in \mathbb{E}_{(\_)} \\ \textit{// randomize i'th child} \\ \hat{C} = (\mathtt{x}_i, \mathtt{y}_i) + [\delta] \cdot H_{\mathsf{other}(\_)} \in \mathbb{E}_{\mathsf{other}(\_)} \end{array} \right\}$$

We can implement this efficiently because the "parent opening" constraints can be directly enforced using a commit-and-prove for Pedersen commitments (Section 5.2). The additional "child opening" requires a single, cheap fixed-based exponentiation explicitly expressed as constraints. We describe an optimized arithmetic circuit for the relation above in the full version.

The following property will be useful for correctness later. It states that the relation above expresses the parent-child relation in a curve tree and that this holds even if we rerandomize children or internal nodes.

Lemma 9 *Consider a set S and a single-level curve tree (one root immediately followed by leaves) built on it. Let $C_{leaf} = (\mathtt{x}_i, \mathtt{y}_i)$ be one of the leaves. Then the above relation $\mathscr{R}^{(\mathsf{single\text{-}level},(\_))}$—for the only existing level $d = 1$—is satisfied for any rerandomization factor δ such that $\hat{C} = C_{leaf} + [\delta] \cdot H_{\mathsf{other}(\_)}$. This property still holds if the root of the tree is rerandomized by some scalar r.*

Proof 14 *This is a straightforward implication of how curve trees are defined. More in detail: In a single-level curve tree, C will be the root and thus constructed with*

$r = 0$. *The first equation will be trivially satisfied by* $\mathbb{x}, \mathbb{y}$ *such that* $((\mathbb{x}_j, \mathbb{y}_j))_j$ *are the leaves (i.e., the children of the root C). The second equation will be satisfied by our assumption on* $\hat{C}$. *We finally observe that we can pick an honestly generated root, rerandomize it by adding* $[r] \cdot H_{(\_)}$ *for a scalar r and use the latter to let the first equation check. This proves the last part of the lemma statement.*

Descending all D Layers of The Tree

So far we discussed proving membership zooming in on a single level of a curve tree. We now want an approach that works for multiple levels. One straightforward method works by providing a separate proof for each level (this would be a proof for the relation in Definition 20). We will do something better instead. We leverage two facts: *i)* that there are two algebraic groups we are working with (depending on the layer parity); *ii)* that we can produce a single proof *at once* and *for multiple layers* "working in the same group". This way we are able to reduce our relation to two proofs only, one for the parents at odd layers and one for parents at even layers.

These two proofs will show respectively two "multi-leveled" relations, one for odd layers and on for even layers. They are defined below.

$$\mathscr{R}^{(\text{evn-levels})} := \left\{ \bigwedge_{j \in \{0,2,\ldots,\mathsf{D}-2\}} \mathscr{R}^{(\mathsf{single\text{-}level},(\text{evn}))} \right\}$$

$$\mathscr{R}^{(\text{odd-levels})} := \left\{ \bigwedge_{j \in \{1,3,\ldots,\mathsf{D}-1\}} \mathscr{R}^{(\mathsf{single\text{-}level},(\text{odd}))} \right\}$$

The witnesses and public statements for these relations are respectively the concatenation of the witnesses and public statements in Definition 20. See also **??**. The full construction is in Figure 5.2.

5.5 Correctness and Security

Theorem 7 *The construction in Figure 5.2 is a transparent select-and-rerandomize (Section 5.3). Its security relies on DLOG (Assumption 2) in* $\mathbb{E}_{(evn)}$ *and* $\mathbb{E}_{(odd)}$ *and the security of Bulletproofs as a NIZKAoK. It has* $O(\sqrt[\mathsf{D}]{n})$ *prover/verifier complexity*[12] *and its proof consists of* $\mathsf{D}-1$ *group elements and two Bulletproofs (each of size* $O(\log \frac{n}{\mathsf{D}})$*).*

Proof 15 *We first observe that, by the DLOG assumption on both curves* $\mathbb{E}_{(odd)}$ *and* $\mathbb{E}_{(evn)}$, *we can use the fact that, by the standard Fiat-Shamir transformation [FS87],* $\mathsf{zk\text{-}BP}[\mathbb{E}_{(odd)}]$ *(resp.* $\mathsf{zk\text{-}BP}[\mathbb{E}_{(evn)}]$*) is a correct, zero-knowledge and extractable NIZK. This will be useful in the remainder of the proof.*

[12] In practice $\mathsf{D} \approx 4$.

$\mathsf{SelRerand.Setup}(1^\lambda) \to \mathsf{pp}$

Sample $\mathbf{G}^{(\text{evn})} \in \mathbb{E}_{(\text{evn})}{}^{N_{\text{urs}}}, H^{(\text{evn})} \in \mathbb{E}_{(\text{evn})}$

Sample $\mathbf{G}^{(\text{odd})} \in \mathbb{E}_{(\text{odd})}{}^{N_{\text{urs}}}, H^{(\text{odd})} \in \mathbb{E}_{(\text{odd})}$

Return all sampled elements as pp

$\mathsf{SelRerand.Comm}(\mathsf{pp}, v_{\text{leaf}} \in \mathbb{F}_{||\mathbb{E}_{(\text{evn})}|}, o \in \mathbb{F}_{||\mathbb{E}_{(\text{evn})}|}) \to C$

$C \leftarrow G_1^{(\text{evn})} \cdot [v_{\text{leaf}}] + H^{(\text{evn})} \cdot [o]$

return $C \in \mathbb{E}_{(\text{evn})}$

$\mathsf{SelRerand.Rerand}(\mathsf{pp}, C \in \mathbb{E}_{(\text{evn})}, r \in \mathbb{F}_{||\mathbb{E}_{(\text{evn})}|}) \to \hat{C}$

$\hat{C} \leftarrow C + H^{(\text{evn})} \cdot [r]$

return $\hat{C} \in \mathbb{E}_{(\text{evn})}$

$\mathsf{SelRerand.Accum}(\mathsf{pp}, S' = \{C_1, \ldots, C_n\}) \to \mathsf{rt}$

Return rt, root of a tree computed on S' as by Definition 19

$\mathsf{SelRerand}.\mathscr{P}(\mathsf{pp}, S, C_{\text{leaf}}, r^{(\mathsf{D})})$

Reconstruct tree from S; let rt be its root

Let $C^{(0)}, \ldots, C^{(\mathsf{D})}$ be the path elements to C_{leaf} in the tree

(with $C^{(0)}$ corresponding to rt, $C^{(\mathsf{D})} = C_{\text{leaf}}$)

Let $\hat{C}^{(0)} := \mathsf{rt}$ and $r^{(0)} := 0$

for $k = 1, \ldots, \mathsf{D}/2$ **do**

$j \leftarrow 2k-1$// $j = 1, 3, \ldots$

$j' \leftarrow 2(k-1)$// $j' = 0, 2, \ldots$

Sample $r^{(j)} \xleftarrow{\$} \mathbb{F}_{|\mathbb{E}_{(\text{odd})}|}$

if $j' < \mathsf{D}$ **then** Sample $r^{(j')} \xleftarrow{\$} \mathbb{F}_{|\mathbb{E}_{(\text{evn})}|}$

$\hat{C}^{(j)} \leftarrow C^{(j)} + \left[r^{(j)}\right] \cdot H_{(\text{odd})}$

$\hat{C}^{(j')} \leftarrow C^{(j')} + \left[r^{(j')}\right] \cdot H_{(\text{evn})}$

endfor

$\pi_{(\text{evn})} \leftarrow \mathsf{zk\text{-}BP}[\mathbb{E}_{(\text{evn})}].\mathsf{Prove}\left(\mathsf{pp}, \mathscr{R}^{(\text{evn-levels})}, \mathsf{x}_{(\text{evn})}, \mathsf{w}_{(\text{evn})}\right)$

$\pi_{(\text{odd})} \leftarrow \mathsf{zk\text{-}BP}[\mathbb{E}_{(\text{odd})}].\mathsf{Prove}\left(\mathsf{pp}, \mathscr{R}^{(\text{odd-levels})}, \mathsf{x}_{(\text{odd})}, \mathsf{w}_{(\text{odd})}\right)$

Return $\pi^* := \left(\hat{C}^{(1)}, \ldots, \hat{C}^{(\mathsf{D}-1)}, \pi_{(\text{evn})}, \pi_{(\text{odd})}\right)$

$\mathsf{SelRerand}.\mathscr{V}(\mathsf{pp}, \mathsf{rt}, \hat{C}_{\text{leaf}}, \pi^*)$

Parse π^* as $\left(\hat{C}^{(1)}, \ldots, \hat{C}^{(\mathsf{D}-1)}, \pi_{(\text{evn})}, \pi_{(\text{odd})}\right)$

Let $\hat{C}^{(\mathsf{D})} := \hat{C}_{\text{leaf}}$

Let $\hat{C}^{(0)} := \mathsf{rt}$

$b_{(\text{evn})} \leftarrow \mathsf{zk\text{-}BP}[\mathbb{E}_{(\text{evn})}].\mathsf{VerProof}\left(\mathsf{pp}, \mathscr{R}^{(\text{evn-levels})}, \mathsf{x}_{(\text{evn})}, \pi_{(\text{evn})}\right)$

$b_{(\text{odd})} \leftarrow \mathsf{zk\text{-}BP}[\mathbb{E}_{(\text{odd})}].\mathsf{VerProof}\left(\mathsf{pp}, \mathscr{R}^{(\text{odd-levels})}, \mathsf{x}_{(\text{odd})}, \pi_{(\text{odd})}\right)$

Accept iff $b_{(\text{evn})} \wedge b_{(\text{odd})} = 1$

Figure 5.2: Construction of Curve Tree Select-and-Rerandomize for a set of size n, branching factor ℓ, depth D (which we assume to be even), on cycle $(\mathbb{E}_{(\text{evn})}, \mathbb{E}_{(\text{odd})})$.

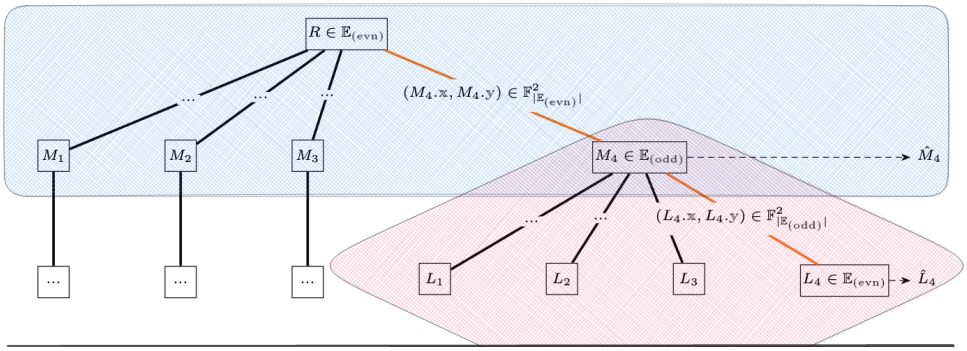

Figure 5.3: Illustrating proving select-and-rerandomize for a tree with $\mathsf{D} = 2$ and $\ell = 4$. Letters R, M, L hint respectively to commitments to root, a "middle" and "lower" layer respectively. The textured box and diamond areas denote the relation proven through Bulletproofs (on different curves, hence the different color). The dashed arrows going towards the right denote rerandomization.

Correctness. *Correctness of rerandomization is immediate: we are using standard Pedersen as a commitment, which is rerandomizable. That is if* $C = G_1^{(evn)} \cdot [v_{leaf}] + H^{(evn)} \cdot [o]$ *then its rerandomization by* r *is* $\hat{C} = C + H^{(evn)} \cdot [r] = G_1^{(evn)} \cdot [v_{leaf}] + H^{(evn)} \cdot [o + r]$.

To argue Select-and-Rerandomize correctness we will invoke these facts: that the output of Comm*—i.e., leaves—are rerandomizable objects (observation from previous paragraph), the fact that internal nodes are rerandomizable (Lemma 8) and finally the correctness of Bulletproofs as NIZK. We can use the above to observe that, for an honestly generated commitment to a set, the honest prover will reconstruct a path, rerandomize each elements and then prove a conjunction of the level equation (*$\mathscr{R}^{(\mathsf{single\text{-}level},(\_))}$*). We can invoke correctness of Bulletproofs if its prover is invoked with a statement satisfying those equations (see Lemma 9). Observing that a conjunction of satisfiable* $\mathscr{R}^{(\mathsf{single\text{-}level},(\_))}$*-s is satisfiable (with corresponding witnesses) concludes the correctness proof.*

Hiding and Zero-knowledge. *Hiding is immediate from properties of Pedersen commitments. We describe a simulator* $\mathscr{S}$ *for the zero-knowledge which outputs* π^* *consisting of :* $\hat{C}^{(1)}, \ldots, \hat{C}^{(\mathsf{D}-1)}$ *fresh commitments to dummy values;* $\pi_{(evn)}$ *and* $\pi_{(odd)}$ *outputs of the respective simulators for the Bulletproofs NIZK on the respective relations. Notice that—by the definition of the game for select-and-rerandomize zero-knowledge and Lemma 9—the Bulletproofs simulators are invoked on true statements, crucially. To argue indistinguishability of the output of our simulator from that of the honest prover, we can just apply a hybrid argument where we invoke hiding of commitments and zero-knowledge of the underlying Bulletproofs.*

Select-and-Rerandomize Binding.

For sake of clarity and simplicity of notation, we first show our proof for the two-level case $\mathsf{D} = 2$ *and then describe how it generalizes.*

The verifier will then receive the following (see also definition of $\pi^\star$ *in Figure 5.2 for context as well as Figure 5.3 for visual cues and an example):*

- *A rerandomized commitment to the leaf* C_{leaf}
- *A proof* π^* *consisting of: 1. a rerandomized commitment* $\hat{C}_{mid}$ *to the intermediate layer (*$\hat{C}^{(1)}$ *in Figure 5.2); 2. an "upper-level" proof* $\pi_\uparrow$*, "linking" root and mid layer; 3. a "lower-level" proof* $\pi_\downarrow$*, "linking" mid and leaf layer.*

As in the definition of binding (Section 5.3), we denote by $\hat{v}$ *a malicious value not in the honestly generated set (but which the adversary "will claim" it's in the set).*
We mark in blue elements that are extracted from the proofs.

Step 1. *Apply knowledge-soundness to extract from the upper proof:*

$$\begin{aligned}\hat{C}_{root} =& \langle \dots \mathbb{x}(C_{mid}) \dots, \mathbf{G}^{\mathsf{x}}_{(\_)} \rangle \\ &+ \langle \dots \mathbb{y}(C_{mid}) \dots, \mathbf{G}^{\mathsf{y}}_{(\_)} \rangle + [r_{root}] \cdot H_{(\_)}\end{aligned} \tag{5.2}$$

$$\hat{C}_{mid} = C_{mid} + [\delta_{mid}] \cdot H_{\mathsf{other}(\_)} \tag{5.3}$$

Observation a). *We observe that above that the extracted* C_{mid} *will be the same as in the honest construction step of the tree (w.l.o.g. we can ignore the specific index on the path for it—this holds for all indices). If this were not the case we would be violating Lemma 8:* C_{root} *is an internal node of the tree and so it is a binding commitment to its children (see statement of Lemma 8). This observation will be useful later since we know the discrete logarithm of* C_{mid} *in* $\mathbf{G}^{\mathsf{x}}_{\mathsf{other}(\_)}, \mathbf{G}^{\mathsf{y}}_{\mathsf{other}(\_)}, H_{\mathsf{other}(\_)}$.

Step 2. *Apply knowledge-soundness to extract from the lower proof:*

$$\begin{aligned}\hat{C}_{mid} =& \langle \dots \mathbb{x}(C_{leaf}) \dots, \mathbf{G}^{\mathsf{x}}_{\mathsf{other}(\_)} \rangle \\ &+ \langle \dots \mathbb{y}(C_{leaf}) \dots, \mathbf{G}^{\mathsf{y}}_{\mathsf{other}(\_)} \rangle + [r_{mid}] \cdot H_{\mathsf{other}(\_)}\end{aligned} \tag{5.4}$$

$$\hat{C}_{leaf} = C_{leaf} + [\delta_{leaf}] \cdot H_{(\_)} \tag{5.5}$$

In addition to the group elements above, we will also extract, i^**, the index* C_{leaf} *refers to (see first witness in Definition 20).* **Observation b).** *Because the adversary is successful in the binding experiment (through some claimed* $\hat{v} \notin S = \{v_i\}_i$*), we can conclude that* i^* *such that* $C_{leaf} \neq \mathsf{Comm}(v_{i^*}, o_{i^*})$*. (Otherwise we would have* $\mathsf{Comm}(v_{i^*}, o_{i^*}) + [\delta_{leaf}] \cdot H_{(\_)} = \hat{C}_{leaf} = \mathsf{Comm}(\hat{v}, \hat{o})$ *which would break DLOG) This is equivalent to saying that* $\mathbb{x}(C_{leaf}) \neq \mathbb{x}_{i^*}$ *or* $\mathbb{y}(C_{leaf}) \neq \mathbb{y}_{i^*}$*, where* $\mathbb{x}_{i^*} := \mathbb{x}(\mathsf{Comm}(v_{i^*}, o_{i^*})), \mathbb{y}_{i^*} := \mathbb{y}(\mathsf{Comm}(v_{i^*}, o_{i^*}))$.

Step 3. *Combine equations Equation* (5.3) *and Equation* (5.4)*: Now, combining the equations, we can observe that:*

$$\begin{aligned}&[r_{mid} - \delta_{mid}] \cdot H_{\mathsf{other}(\_)} - C_{mid} + \\ &\langle \dots \mathbb{x}(C_{leaf}) \dots, \mathbf{G}^{\mathsf{x}}_{\mathsf{other}(\_)} \rangle + \langle \dots \mathbb{y}(C_{leaf}) \dots, \mathbf{G}^{\mathsf{y}}_{\mathsf{other}(\_)} \rangle = 0\end{aligned}$$

This allows an adversary to break DLOG (Assumption 2) by using the following facts. As we observed (obs. (a)), C_{mid} is the same as in the honest tree construction, which implies its discrete logarithms can be derived knowing the original honest set. If $\mathbb{x}(C_{leaf}) \neq \mathbb{x}_{i^*}$, the adversary can then break DLOG for the generator $G^{\mathsf{x}}_{i^*,\mathsf{other}(\_)}$. This becomes clear when rewriting the equation above like this:

$$G^{\mathsf{x}}_{i^*,\mathsf{other}(\_)} = \Big(\underbrace{\mathbb{x}_{i^*} - \mathbb{x}(C_{leaf})}_{\neq 0}\Big)^{-1} \cdot \Big([r_{mid} - \delta_{mid}] \cdot H_{\mathsf{other}(\_)} + \langle \mathbb{x}^{(leaf)}_{\neq i^*}, \mathbf{G}^{\mathsf{x}}_{\neq i^*,\mathsf{other}(\_)} \rangle + \langle \ldots \mathbb{y}(C_{leaf}) \ldots, \mathbf{G}^{\mathsf{y}}_{\mathsf{other}(\_)} \rangle \Big)$$

where $\mathbb{x}^{(leaf)}_{\neq i^*} := \big(\mathbb{x}(\mathsf{Comm}(v_i, o_i))\big)_{i \neq i^*}$. If $\mathbb{y}(C_{leaf}) \neq \mathbb{y}_{i^*}$, we can modify the above accordingly to apply to $\mathbb{y}$. This concludes the proof.

To generalize the proof to $\mathsf{D} \geq 2$. *First, we recursively apply Step 1 and observation (a), i.e., we repeatedly apply Lemma 8 to argue that is the same as in the honestly constructed tree for each internal node C_{mid} on the path. Then, as we did above, we apply step 2 and step 3 for the last two layers, as well as observation b). (Notice that, in order to extract the equations, we will still use two proofs but now each of them will allow us to extract multiple levels. There are still only two proofs—even and odd—but now they refer to multiple disjoint levels of the tree instead of just two).*

5.6 Final Construction: Curve Trees with Compressed Points

In this section we describe some optimizations we employ in our final construction. Our initial observation is that a curve tree (as defined in Definition 19) uses both $\mathbb{x}$ and $\mathbb{y}$ coordinates to represent a node (leaf or internal). This requires 2ℓ generators at each level. The factor 2 will become a cost at commitment, proving and verification time, as well in proof size. Here we discuss how to remove this factor.

The starting point of our idea are folklore approaches to point compression which rely on encoding a point through the $\mathbb{x}$ coordinate. We need to take extra care though. Where we need to take extra care is in: a) making sure, through appropriate checks. that a malicious prover cannot exploit this compression; b) making sure the latter checks are efficient constraints-wise when we prove/verify them in zero-knowledge. In order to do this we exploit the fact that the leaves in the tree are agreed on publicly (we remind that in our model as well in confidential transactions, the whole set of points is public; the item we prove membership on is hidden). This way, we can make sure at commitment time that each leaf is represented through pairs of points of a certain form. We call these points *permissible*. We modify our definition of curve trees to explicitly take compression and permissibility into account (Definition 21). To efficiently prove/verify this we rely on 2-universal hash functions (see rest of this section and Equation (5.6)). Their algebraic nature allows us to not to employ bit decomposition. As a consequence, these techniques have nearly no impact on any additional complexity of the relation proved in zero-knowledge.

When we to plug in these additional tricks, our construction (Figure 5.2) stays essentially the same: we can describe its changes in a modular fashion (see Section 5.6). The same holds for security and correctness proofs.

Point Compression and Permissible Points

In order to reduce the number of exponentiations during commitment and the size of the witness we rely on committing only to the $\mathbb{x}$-coordinate of children node. To guarantee that our construction remains binding we ensure that only one of $(\mathbb{x},\mathbb{y})$ and $(\mathbb{x},-\mathbb{y})$ is "allowed". One common choice is to take the numerically smallest between $\mathbb{y}$ and $-\mathbb{y}$, or discriminate based upon the parity (even/odd) over $\mathbb{Z}$, however neither of these constraints can be efficiently expressed as an arithmetic circuit; instead we use a universal hash function (which does not require bit decomposition). Let $S(v) = 1$ iff. $v \in \mathbb{F}$ is a quadratic residue (i.e. there exists $w \in \mathbb{F}$ st. $w^2 = v$) and $S(v) = 0$ otherwise. Now consider the following family of 2-universal hash functions from any field to $\{0,1\}$:

$$\mathscr{U}_{\alpha,\beta}(v) : \mathbb{F} \to \{0,1\} \quad \mathscr{U}_{\alpha,\beta}(v) \mapsto S(\alpha \cdot v + \beta) \tag{5.6}$$

Observe that the constraint $\mathscr{U}_{\alpha,\beta}(v) = 1$ can be enforced using a circuit with multiplicative complexity 1, showing $\{(w) : w^2 = (\alpha \cdot v + \beta)\}$. We exploit this to efficiently define a set of "permissible points" on $\mathbb{E}$:

$$\mathscr{P}_{\mathbb{E}} = \{(\mathbb{x},\mathbb{y}) \mid (\mathbb{x},\mathbb{y}) \in \mathbb{E}(\mathbb{F}_p) \wedge \mathscr{U}_{\alpha,\beta}(\mathbb{y}) = 1 \wedge \mathscr{U}_{\alpha,\beta}(-\mathbb{y}) = 0\}$$

Note that $1/4$ of the points on $\mathbb{E}$ are permissible and any $(\mathbb{x},\mathbb{y}) \in \mathscr{P}_{\mathbb{E}}$ is uniquely defined by its $\mathbb{x}$-coordinate – this is the case for any finite field of characteristic $\notin \{2,3\}$.

We make sure at commitment time that nodes are converted to permissible points by adding appropriate randomness. This is formalized in the supplementary material in **??** in the procedure AsPermissible as well in the "compressed points" definition of curve trees (Definition 21), which invokes it. In expectation, procedure AsPermissible requires 4 curve additions and 8 square roots.

Curve Trees with Compressed Points

The following definition simply adapts a curve tree to the setting where leaves are required to be permissible and internal children nodes are compressed through AsPermissible before committing to them in their parent.

Definition 21 (Curve Trees with compressed points) *A Curve Tree with compressed points follows the same basic inductive definition as Definition 19, but with the following differences: first, the tree is also parametrized by two permissible sets $\mathscr{P}_{(evn)} \subseteq \mathbb{E}_{(evn)}$ and $\mathscr{P}_{(odd)} \subseteq \mathbb{E}_{(odd)}$. Second, Equation* (5.1) *(root label C of an internal node) becomes*

$$C = \langle \langle \mathbb{x} \rangle', \mathbf{G}^{\mathbb{x}}_{(\_)} \rangle \in \mathbb{E}_{(\_)} \tag{5.7}$$

where for each $i \in [\ell]$, $\mathbb{x}'_i$ is such that $(\mathbb{x}'_i, \dots) \leftarrow \mathsf{AsPermissible}_{\mathbb{E}_{(\_)}}(\mathbb{x}_i, \mathbb{y}_i)$, and $(\mathbb{x}_i, \mathbb{y}_i)$ are as in Equation (5.1)*. Third, leaves are required to be permissible.*

Since the definition above makes a tree only out of permissible points[13] this gives a "decompression" that is unique. This in turn reduces the the complexity single-level relations. We thus define a new optimized relation $\mathscr{R}^{(\mathsf{single\text{-}level}^\star,(\_))}$:

Definition 22 (Optimized single-level relation)

$$\mathscr{R}^{(\mathsf{single\text{-}level}^\star,(\_))} := \left\{ \begin{pmatrix} i, r, \delta, \\ \langle \mathbb{x} \rangle, \mathbb{y} \end{pmatrix} : \begin{array}{c} C = \langle [\langle \mathbb{x} \rangle], \mathbf{G}^{\mathsf{x}}_{(\_)} \rangle \\ + [r] \cdot H_{(\_)} \\ \wedge (\mathbb{x}_i, \mathbb{y}) \in \mathscr{P}_{\mathsf{other}(\_)} \\ \wedge \hat{C} = (\mathbb{x}_i, \mathbb{y}) + [\delta] \cdot H_{\mathsf{other}(\_)} \end{array} \right\}$$

Note that the constraint $(\mathbb{x}_i, \mathbb{y}) \in \mathscr{P}_{\mathsf{other}(\_)}$ only requires a check that $(\mathbb{x}_i, \mathbb{y}) \in \mathbb{E}_{\mathsf{other}(\_)}$ in addition to $\mathscr{U}_{\alpha,\beta}(\mathbb{y}) = 1$.

Adapting Construction in Figure 5.2 to Compressed Points

Our final construction essentially remains the same as in Figure 5.2 with two exceptions.

- In order to accumulate a set ($\mathsf{SelRerand.Accum}$) we generate a root through the procedure derived from Definition 21 instead of the one for Definition 19.
- The proofs $\pi_{(\text{evn})}$ and $\pi_{(\text{odd})}$ are for slightly different relations: they prove/verify relations for $\mathscr{R}^{(\text{evn-levels})}$ and $\mathscr{R}^{(\text{odd-levels})}$ but defined in terms of $\mathscr{R}^{(\mathsf{single\text{-}level}^\star,(\_))}$ from Definition 22 (instead of Definition 20).

This variant construction is also correct and secure:

Theorem 8 *The variant of the construction of Figure 5.2 described in this section is a transparent select-and-rerandomize primitive (under the same assumptions as in Theorem 7).*

The proof for theorem above follows the same blueprint as the one in Theorem 7. Zero-knowledge/hiding is trivially untouched by the changes in the construction. Binding is clearly still guaranteed since the relation we prove ($\mathscr{R}^{(\mathsf{single\text{-}level}^\star,(\_))}$) is now *stricter* than the one in $\mathscr{R}^{(\mathsf{single\text{-}level},(\_))}$. Observing correctness only requires observing that a variant of Lemma 9 also holds (easily) for definition Definition 21.

[13] In case of our anonymous cryptocurrency application, this is enforced by the network of block validators: as a condition for a transaction being valid.

5.7 𝕍Cash: Transparent and Efficient Anonymous Payment System

In this section we informally describe our anonymous payment system, which we dub 𝕍Cash. The techniques and model here follow mostly prior work.

Model

A formal description of our model is in the appendix in **??**. The ideal functionality in the appendix describes the simple expected behavior of an anonymous payment system: parties hold values; they can transfer part of these values to other parties; an adversary can observe transactions but it cannot tamper them or learn anything about the sender/receiver/value of the transaction. This functionality, in particular, supports the largest possible anonymity set at every transaction like ZCash.

A high-level view of our protocol

The flow of our protocol roughly follows known blueprints. We refer the reader to, e.g., the technical overview and Section 3 in [CHA21, CHA22c] for further background.

Intuition about our construction. At any given moment in time, each party holds a certain number of coins[14]. Coins are the fundamental concept in a transaction. During a transaction we *pour* a certain amount from user to user by using two (unspent) input coins and producing two new output coins.

Each user is also holding a public state (the ledger $\mathscr{L}$) roughly containing all the transfers occurred so far. Through the state any user can verify the validity of each transfer. In addition to the public state, users hold a private state containing information as: the aforementioned auxiliary information to spend their coins, signing keys, etc.

In order to implement an anonymous payment system we thus require four algorithms that are run locally by each party in the system:

Setup The setup algorithm produces the initial parameters of the system. We emphasize that it does not require being run by a trusted setup.

Pour A sender $\mathscr{S}$ can "pour" the value of two *input coins* into two new *output coins* nullifying the input ones. The recipients of the two new coins can be distinct. It is possible for $\mathscr{S}$ itself to be one or both of the recipients. We require that the total value of input and output coins is the same. The algorithm Pour has two outputs: a new transaction that is publicly broadcasted and a private auxiliary opening that is sent to the respective recipients of the new output coins.

Verify A verifying algorithm allows any party to check a transaction is valid. It takes as a input the public parameters and the public state observed so far.

[14] "Holding" a coin requires knowing a certain secret key associated to the user. In this section we ignore the aspect of registering with a new key to the system, but we stress it is straightforward to add.

Process By a processing algorithm parties can update their public and private state after observing a transaction.

Our protocol in more detail

We describe our protocol in Figure 5.5 and in the rest of this section.

A transaction consist of the creation of output coins from input coins. A coin roughly consists of a commitment to its amount and other information that ensures it will be used only once and by its intended recipient. For a transaction to be valid it must be the case that:

1. Output coins are in an appropriate non-negative range (we want to *give* money and not take it in a transaction). This corresponds to the Mint in Figure 5.4.

2. Input coins "exist" and are valid themselves. This corresponds to the Spend in Figure 5.4.

3. The total value of input and output coins is the same. This is handled by π_{bal} in Figure 5.5.

We use zero-knowledge proofs to ensure the above. The first and third property can be ensured respectively by range proofs and homomorphic properties of Pedersen commitments & proving knowledge of appropriate discrete logarithms. The second property is where we use our select-and-rerandomize constructions from the previous sections: all coins are stored in an accumulator (a Curve Tree) and whenever they aim to spend an input coin, they can select-and-rerandomize it obtaining a rerandomized version of that input coin. This is included in the transaction together with a proof that it refers to the rerandomization of something existing in the accumulator.

Further details on our building blocks follow.

Breakdown of public parameters: • public parameters for SelRerand • urs (uniform reference string) for zero-knowledge • generators $\left(G_v, G_t, \hat{H}\right)$ for Pedersen commitments whose semantics we explain below.

Structure of a coin: A coin is a Pedersen commitment to: 1) the amount v transferred through the coin; 2) the tag/nullifier t, i.e. the (rerandomized) public key of the recipient. Hence each coin c is of the form $\mathsf{c} = [v] \cdot G_v + [t] \cdot G_t + [r] \cdot \hat{H}$ where r is the randomness we use for masking the polynomial.

Additional cryptographic primitives:

- Digital signatures with rerandomizable keys (see, e.g., [FKM$^+$16]). The key property we require is that we can rerandomize a public key and correspondingly update a signing key. We use this feature in Mint in Figure 5.4.

- Non-Interactive zero-knowledge for different relations:

  - Relation R_{dlog}, which shows knowledge of discrete logarithm for given generators for an input group element c. We use this relation to show zero-balance among input and output coins and to show knowledge of values in the input coins. Whenever we use relation R_{dlog} we also explicitly describe with respect to what tuple of generators. For instance, if we write $R_{\text{dlog}}\left(G_t, \hat{H}\right)$ it means that we are showing knowledge of (t,r) so that a certain commitment equals $[t]\cdot G_t + [r]\cdot \hat{H}$. The last example is instructive in one more way: that relation is equivalent to stating that the "transferred value v" inside a certain commitment (a coin) is zero. We use this fact to assert that the values of input and output coins is balanced overall.
  - Relation $R_{\geq 0}$, which shows knowledge of discrete logarithms for a coin plus that the value of the coin is in a positive range. That is it shows knowledge of (v,t,r) such that $\mathsf{c} = [v]\cdot G_v + [t]\cdot G_t + [r]\cdot \hat{H} \wedge v \in [0, 2^{64})$.

- We denote by $\mathscr{H}_{\mathbb{F}}$ a collision resistant hash function mapping group elements—the public keys of the users—to the appropriate scalar field $\mathbb{F}$. We use this hash function to be able to commit to the public keys as tags. Notice that we do not need to prove this hash function in zero-knowledge.

Other components of public state (i.e., the ledger):

- Set of coins S_{coins} (from which it is possible to compute the corresponding Curve Tree root $\mathsf{rt}_{\text{coins}}$)
- Set of seen "tags". Tags are (rerandomized) public keys of recipients. These are revealed every time an input coin is spent. We stress that they are unlinkable to the actual input coins they refer to because of the select-and-rerandomize proof.

We describe setup and processing algorithm at a very high level since they are almost immediate from the rest of the protocol. The setup algorithm generates all the public parameters described above; it should also provide an initial distribution of coins to users (the mechanism of this initial distribution is unimportant for our focus). The processing algorithm consists in keeping the public state above up to date after each (valid) transaction. It simply updates the set of coins with the new observed output coins and the set of seen tags with those in the latest transaction.

Remark 9 (Optimizations) *The construction in Figure 5.5 shows a separate proof for each of the relations of interest. This is for clarity only. Our final construction produces a single Bulletproof proof whenever possible, thus avoiding a linear overhead in the number of relations. The final numbers are those stated in Section 5.8 and consist of two Bulletproofs lying on two different curves.*

Remark 10 (Full security through efficient PRF) *The scheme in Figure 5.5 is a slightly simplified version of our final protocol for didactic purposes. The simplification has to do with how we generate new tags ($G^{(j)}_{nll,out}$). The scheme in the figure, as it*

is, has an additional leakage: a party $\mathscr{S}$ sending a transaction tx *to a party $\mathscr{R}$ can learn when R will spend the coins received in* tx *(but not to whom). Only sender S can infer this. Additionally the scheme suffers from "Faerie's Gold Attack", which enables an adversary to create two distinct transactions of which only one can be spent by the honest receiver. Our final scheme mitigates both of these issues using a PRF. This solution is similar to that used in Zcash. Differently than Zcash we can exploit a more efficient way to prove the PRF computation—thanks to our choice of PRF and groups. However, in order to avoid bloating the circuit to be proven in ZK, we use a "commit-and-prove friendly" PRF with bounded-query security. The fact that we need to require this bound beforehand is not a problem since we can use a bound on the number of transactions we expect in the system (e.g. a very conservative bound of 2^{32} transactions per-user). We give a concrete instantiation based on Diffie-Hellman Inversion Assumption (DHI) using a PRF is based upon Dodis and Yampolskiy [DY05] where $\mathsf{PRF}_K(x) = \left[(K+x)^{-1}\right] \cdot G$. Security of this extensions follows from the well-studied Diffie-Hellman Inversion (DHI) assumption [MSK02]. More details are in* **??**. ***NB:*** *differently from [DY05], our instantiation group is pairing-free and thus we can instead obtain an evaluation proof through an additional opening of a group element in Bulletproof (alternatively one could use a Sigma-protocol).*

5.8 Implementation and Evaluation

We implement select-and-rerandomize and $\mathbb{V}$Cash in Rust on top of the `dalek` Bulletproofs library[15]. The Bulletproof implementation has been extended with support for vector commitmentsand elliptic curves implemented using the arkworks[16] curve traits.

CODE. All our code is available and released as open source at

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

EXPERIMENTAL SETTING AND INSTANTIATIONS. Our benchmarks were run on a C6i.2xlarge[17] instance with 8 vCPUs, which corresponds to 4 physical cores on an Intel Xeon 8375C processor with 2.9 GHz clock speed[18]. When possible (and unless otherwise explicitly specified) we have benchmarked alternative schemes on the same hardware. We use Curve Trees of even depth D in our evaluation and instantiate the two underlying elliptic curves through both those in the Pasta cycle [Hop20] and the `secp256k1` / `secq256k1` cycle. We use Schnorr signatures for $\mathbb{V}$Cash.

[15] `https://github.com/dalek-cryptography/bulletproofs`

[16] `https://github.com/arkworks-rs`

[17] `https://aws.amazon.com/ec2/instance-types/c6i/`

[18] While we tabulate only results for this architecture, we also performed benchmarks on a common laptop.

$\mathsf{Spend}\left(\mathsf{aux}_{\text{in}}^{(j)}\right)$

// Reconstruct input coin

Parse $\mathsf{aux}_{\text{in}}^{(j)}$ as $\left(v_{\text{in}}^{(j)}, \mathscr{S}_{\text{rr}}^{(j)}, r_{\text{in}}^{(j)}\right)$

$G_{\text{nll,in}}^{(j)} \leftarrow \mathscr{H}_{\mathbb{F}}\left(\mathscr{S}_{\text{rr}}^{(j)}\right) \cdot G_t$ // reconstruct input tag

$\mathsf{c}_{\text{in}}^{(j)} \leftarrow \left[v_{\text{in}}^{(j)}\right] \cdot G_v + G_{\text{nll,in}}^{(j)} + \left[r_{\text{in}}^{(j)}\right] \cdot \hat{H}$ // reconstruct coin

// Select-and-Rerandomize input coin

$\left(\mathsf{c}_{\text{rr}}^{(j)}, \pi_{\text{SR}}(j), r_{\text{rr}}^{(j)}\right) \leftarrow \mathsf{SelRerand}.\mathscr{P}\left(\mathsf{pp}_{\text{SR}}, S_{\text{coins}}, \mathsf{c}_{\text{in}}^{(j)}\right)$

// Prove knowledge of opening of input coin

$\pi_{\text{spnd}}^{(j)} \leftarrow \mathsf{ZK.Prove}\left(\mathsf{urs}, R_{\text{dlog}}\left(G_v, \hat{H}\right), \mathsf{c}_{\text{rr}}^{(j)} - G_{\text{nll,in}}^{(j)}; \mathsf{aux}_{\text{in}}^{(j)}, r_{\text{rr}}^{(j)}\right)$

$\mathsf{Mint}\left(\mathscr{R}^{(j)}, v_{\text{out}}^{(j)}\right)$

$r_{\text{out}}^{(j)} \xleftarrow{\$} \mathbb{F}$ // to mask coin

$r_{\text{pk}}^{(j)} \xleftarrow{\$} \mathbb{F}$ // to rerandomize pk

$\mathscr{R}_{\text{rr}}^{(j)} \leftarrow \left[r_{\text{pk}}^{(j)}\right] \cdot \mathscr{R}^{(j)}$ // rerandomized pk

$G_{\text{nll,out}}^{(j)} \leftarrow \mathscr{H}_{\mathbb{F}}\left(\mathscr{R}_{\text{rr}}^{(j)}\right) \cdot G_t$ // make output tag

$\mathsf{c}_{\text{out}}^{(j)} \leftarrow \left[v_{\text{out}}^{(j)}\right] \cdot G_v + G_{\text{nll,out}}^{(j)} + \left[r_{\text{out}}^{(j)}\right] \cdot \hat{H}$ // make coin

$\mathsf{aux}_{\text{out}}^{(j)} \leftarrow \left(v_{\text{out}}^{(j)}, \mathscr{R}_{\text{rr}}^{(j)}, r_{\text{out}}^{(j)}\right)$ // opening of coin

// Proves value of coin ≥ 0

$\pi_{\geq 0}^{(j)} \leftarrow \mathsf{ZK.Prove}\left(\mathsf{urs}, R_{\geq 0}, \mathsf{c}_{\text{out}}^{(j)}; \mathsf{aux}_{\text{out}}^{(j)}, \mathscr{R}^{(j)}\right)$

Figure 5.4: Auxiliary algorithms for algorithm Pour. We assume all variables have the same scope as Pour.

Zero-Knowledge for Set-Membership

The results in Table 5.1 summarize the efficiency of our select and rerandomize scheme (Section 5.3) using the final construction in Section 5.6 for different set sizes—modest, medium and large. Given a choice of parameters—the branching factor ℓ and (even) depth D—the total number of constraints to prove in zero-knowledge amounts to $\mathsf{D}(912+\ell-1)$ (half per even/odd layers respectively). We heuristically choose the set size ($|S| = \ell^{\mathsf{D}}$) in order to optimize the running time by obtaining a number of constraints which "does not overflow" powers of two if possible. This is illustrated by the benchmarks for sets of size 2^{32} and 2^{40}: despite the gap between the set sizes they show similar performance.

If only proofs of membership of field elements are needed, these can be achieved by using the select and rerandomize scheme on vector commitments of ℓ' elements

$\mathsf{Pour}\left(\mathsf{pp}, \mathsf{st}_{\mathscr{S}}, \left(\mathscr{S}^{(j)}, \mathsf{aux}_{\text{in}}^{(j)}, \mathscr{R}^{(j)}, v_{\text{out}}^{(j)}\right)_{j\in[2]}\right)$

// Create output coins

for $j \in [2]$:

$\mathsf{Mint}\left(\mathscr{R}^{(j)}, v_{\text{out}}^{(j)}\right)$

// Show we are using existing coins

for $j \in [2]$:

$\mathsf{Spend}\left(\mathsf{aux}_{\text{in}}^{(j)}\right)$

// Show that $v_{\text{in}}^{(1)} + v_{\text{in}}^{(2)} = v_{\text{out}}^{(1)} + v_{\text{out}}^{(2)}$

$\mathsf{c}_{\text{bal}} \leftarrow \mathsf{c}_{\text{out}}^{(1)} + \mathsf{c}_{\text{out}}^{(2)} - \mathsf{c}_{\text{rr}}^{(1)} - \mathsf{c}_{\text{rr}}^{(2)}$

$\pi_{\text{bal}} \leftarrow \mathsf{ZK.Prove}\big(\mathsf{urs}, R_{\text{dlog}}\left(G_t, \hat{H}\right), \mathsf{c}_{\text{bal}};$
$\mathsf{aux}_{\text{in}}^{(j)}, r_{\text{rr}}^{(j)}, \mathsf{aux}_{\text{out}}^{(j)}, \mathscr{R}^{(j)}\big)$

$\mathsf{tx} := \left(\left(\mathscr{S}_{\text{rr}}^{(j)}, \mathsf{c}_{\text{rr}}^{(j)}, \mathsf{c}_{\text{out}}^{(j)}, \mathscr{S}_{\text{rr}}^{(j)}\right)_{j\in[2]}, \text{proofs } \pi_\star\right)$

Double sign tx with sk-s **for** $\mathscr{S}^{(1)}$ and $\mathscr{S}^{(2)}$

Privately send $\left(\mathsf{aux}_{\text{out}}^{(j)}\right)_{j\in[2]}$; Broadcast tx

$\mathsf{Vfy}\left(\mathsf{pp}, \mathsf{tx} := \left(\left(\mathscr{S}_{\text{rr}}^{(j)}, \mathsf{c}_{\text{rr}}^{(j)}, \mathsf{c}_{\text{out}}^{(j)}, \mathscr{S}_{\text{rr}}^{(j)}\right)_{j\in[2]}, \text{proofs } \pi_\star\right), \mathscr{L}\right)$

for $j \in [2]$:

check $\mathsf{SelRerand.Vfy}\left(\mathsf{pp}_{\text{SR}}, \mathsf{rt}_{\text{coins}}, \mathsf{c}_{\text{rr}}^{(j)}, \pi_{\text{SR}}^{(j)}\right)$

$G_{\text{nll,in}}^{(j)} \leftarrow \mathscr{H}_{\mathbb{F}}\left(\mathscr{S}_{\text{rr}}^{(j)}\right) \cdot G_t$ // reconstruct tags

Reject if $G_{\text{nll,in}}^{(j)}$ has been seen before already

check $\mathsf{ZK.Vfy}\left(\mathsf{urs}, R_{\text{dlog}}\left(G_v, \hat{H}\right), \mathsf{c}_{\text{rr}}^{(j)} - G_{\text{nll,in}}^{(j)}, \pi_{\text{spnd}}^{(j)}\right)$

check $\mathsf{ZK.Vfy}\left(\mathsf{urs}, R_{\geq 0}, \mathsf{c}_{\text{out}}^{(j)}, \pi_{\geq 0}^{(j)}\right)$

$\mathsf{c}_{\text{bal}} \leftarrow \mathsf{c}_{\text{out}}^{(1)} + \mathsf{c}_{\text{out}}^{(2)} - \mathsf{c}_{\text{rr}}^{(1)} - \mathsf{c}_{\text{rr}}^{(2)}$

Check $\mathsf{ZK.Vfy}\left(\mathsf{urs}, R_{\text{dlog}}\left(G_t, \hat{H}\right), \mathsf{c}_{\text{bal}}, \pi_{\text{bal}}\right)$

Verify signatures on tx with public keys for $\mathscr{S}_{\text{rr}}^{(j)}$-s

Accept iff all checks above succeed

Figure 5.5: Pour and Verification algorithms in $\mathbb{V}$Cash.

Curves	(D, ℓ)	Set size	# Constraints	Proof size (kb)	Proving time (s)	Verification time (ms)	Amort. batch verification time (ms)
Pasta	$(2, 1024)$	2^{20}	3870	2.6	0.88	23.17	1.44
	$(4, 256)$	2^{32}	4668	2.9	1.71	39.63	2.35
	$(4, 1024)$	2^{40}	7740	2.9	1.74	40.41	2.73
Secp/Secq	$(2, 1024)$	2^{20}	3870	2.6	0.97	26.81	1.61
	$(4, 256)$	2^{32}	4668	2.9	1.89	47.39	2.64
	$(4, 1024)$	2^{40}	7740	2.9	1.92	48.40	3.02

Table 5.1: Benchmarks of the select and rerandomize primitive with depth D and branching factor ℓ. The amortized batch verification time refers to a batch of 100 proofs.

Scheme	# Constraints	Prove (s)	Verify (ms)	Verify batch (ms)
Curve Trees (Pasta)	3565	1.5	31	1.8
Curve Trees (Secp/Secq)	3565	1.7	37	2
Poseidon 4:1	4515	8.8	651	-
Poseidon 8:1	4180	8.5	825	-

Table 5.2: Benchmarks of accumulators over sets of size 2^{30} based respectively on curves trees and on Merkle trees with Poseidon (**??**). Batch verification time is for the amortized time for a batch of size 100.

obtaining a scheme which uses only $\mathsf{D}(912 + \ell - 1) + (\ell' - 1)$ constraints to show membership of a set with $\ell^{\mathsf{D}} \cdot \ell'$ elements. Using the parameters $\mathsf{D} = 3$, $\ell = 256$, and $\ell' = 64$ we get a direct comparison ($|S| = 2^{30}$) with [GKR$^+$21] in which they use bulletproofs to show membership in Poseidon based Merkle trees with 2^{30} leaves and $\ell = 2$, 4, or 8. The best performing instances in [GKR$^+$21] are using branching factors of 4 and 8 on the ed25519 curve: one results in slightly fewer constraints and faster proving time, while the other verifies faster. The results in Table 5.2 show that the accumulator based on Curve Trees is > 5 times faster at proving and > 20 times faster at verifying compared to the fastest instances of Poseidon-based Merkle trees.

$\mathbb{V}$Cash

Table 5.3 compares $\mathbb{V}$Cash with various anonymous payment systems. When used for batch verification, $\mathbb{V}$Cash outperforms other schemes, sometimes by orders of magnitude (for the same anonymity sets). Non-batched verification time is highly competitive when compared to transparent constructions, but $10\times$ slower than the non-transparent Zcash Sapling (which mainly consists of a few pairing operations). Orchard—the recent transparent version of Zcash based on Halo2 and Pasta (see also **??**—achieves a $5\times$ faster verification time than $\mathbb{V}$Cash. We believe that basing $\mathbb{V}$Cash on a Curve Tree using Halo2 would outperform Orchard. On the other hand this would come at the price, as it is the case for Orchard, of not supporting arbitrary 2-cycles of curves (see Section 5.1). The transaction size in Orchard is roughly

	Anonymity set size	Transparent setup	Tx size (kb)	Proving time (S)	Verification time (ms)	Amort. batch verification time (ms)
Zcash Sapling	2^{32}	✗	2.8	2.38	7	-
Zcash Orchard	2^{32}	✓	7.6	1.77	15	-
𝕍eksel	Any	✗⋆	5.3	0.44	61.88	-
Lelantus	2^{10}	✓	2.7	0.27†	-	6.8†
	2^{14}	✓	3.9	2.35†	-	10.2†
	2^{16}	✓	5.6	4.8†	-	52†
Omniring	2^{10}	✓	1	$\approx$ 1.5‡	$\approx$ 130‡	-
𝕍Cash (Pasta)	2^{20}	✓	3.4	1.76	41.40	2.87
	2^{32}	✓	4	3.43	78.40	4.98
	2^{40}	✓	4	3.48	80.52	5.77
𝕍Cash (Secp/Secq)	2^{20}	✓	3.4	1.95	48.27	3.15
	2^{32}	✓	4	3.80	90.40	5.60
	2^{40}	✓	4	3.86	91.97	6.32

Table 5.3: Benchmarks of 𝕍Cash against other anonymous payment schemes. The 𝕍Cash schemes are instantiated with Curve Trees with the corresponding set size in Table 5.1. The batch verification time is measured as the cost per proof of verifying a batch of 100 proofs. If batch verification is empty, it means it is not available as an option for that specific construction or not possible to estimate from the related work.

⋆ 𝕍eksel only needs setup if using accumulators instantiated with RSA (which provide the smallest tx size), but not for zero-knowledge.

† Lelantus was benchmarked on an Intel i7-4870HQ (4 cores, 2.5GHz).[Jiv19]

‡ Omniring was benchmarked on an Intel i7-7600U (2 cores, 2.8GHz).[LRR+19].

twice as large as in 𝕍Cash. The only other better transaction size among transparent constructions is that of Omniring (we estimate 𝕍Cash to be less than $2\times$ larger for same anonymity sets).

Concretely, a "pour" in 𝕍Cash for two inputs/two outputs and anonymity sets of 2^{32} (like in Zcash) our confidential transactions (𝕍cash) require participants to compute/verify two Bulletproofs proofs of < 5000 constraints each. We can contrast that to another approach supporting large anonymity sets, Zcash Sapling, compared to which our circuit for "spend" transaction is 20x smaller. The cost of the set membership proofs dominate the combined transaction circuit. For instance the 𝕍Cash combined circuit (over both fields) has 9464 constraints of which 9336 are used for the proof of membership and the Orchard action circuit has 2^{11} rows and 40 columns while the membership by itself uses 2^{11} rows and 35 columns.

We remark that, in the table, we only compare to approaches with concretely small transaction size (of a few kilobytes for large enough anonymity sets). Solutions not in the table because of their large transaction size include: the original approach in Zerocoin [MGGR13] (45KB for full security [CHA22c]); Quisquis [FMMO19] (13KB for $|S| = 2^4$); Monero [AJ18] (whose transaction grows linearly with $|S|$ and is already at 1.3KB for $|S| < 2^4$).

5.9 Accumulators

For reference and to make easier a comparison to our primitive in Definition 18, we provide the more standard definition of accumulators [BBF19].

Definition 23 (Accumulator scheme) *An accumulator scheme* Acc *over universe* $\mathscr{U}_\lambda(\mathsf{Acc})$ *(for a security parameter* λ*) consists of PPT algorithms* $\mathsf{Acc} = (\mathsf{Setup}, \mathsf{Accum}, \mathsf{PrvMem}, \mathsf{VfyMem})$ *with the following syntax:*

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$ generates public parameters pp.

$\mathsf{Accum}(\mathsf{pp}, S) \to A$ deterministically computes accumulator A for set $S \subseteq \mathscr{U}_\lambda(\mathsf{Acc})$.

$\mathsf{PrvMem}(\mathsf{pp}, S, x) \to W$ computes witness W that proves x is in accumulated set S.

$\mathsf{VfyMem}(\mathsf{pp}, A, x, W) \to b \in \{0,1\}$ verifies through witness whether x is in the set accumulated in A. We do not require parameter x to be in $\mathscr{U}_\lambda(\mathsf{Acc})$ from the syntax.

Correctness: For any set $S = \{v_i\}_i$, $j^* \in [|S|]$ the following holds

$$\Pr\begin{bmatrix} \mathsf{pp}_{\mathrm{acc}} \leftarrow \mathsf{Acc.Setup}(1^\lambda) \\ A = \mathsf{Acc.Accum}(\mathsf{pp}, S) \ : \\ \pi \leftarrow \mathsf{Acc.PrvMem}(\mathsf{pp}_{\mathrm{acc}}, S, v_{j^*}) \end{bmatrix} = 1$$

Security: For any PPT adversary $\mathscr{A}$ the following holds:

$$\Pr\begin{bmatrix} \mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda) & \\ (S, v', \pi) \leftarrow \mathscr{A}(\mathsf{pp}_{\mathrm{acc}}) \ : & \mathsf{Acc.VfyMem}(\mathsf{pp}, A, v', \pi) \\ A = \mathsf{Acc.Accum}(\mathsf{pp}, S) & \wedge\ v' \notin S \end{bmatrix}$$
$$\leq \mathsf{negl}(\lambda)$$

Chapter 6

Secure Multiparty Computation with Free Branching

Aarushi Goel, Mathias Hall-Andersen, Aditya Hedge, Abhishek Jain.

Orignally published at Eurocrypt 2022.

6.1 Introduction

Secure multiparty computation (MPC) [Yao86, GMW87, CCD88, BGW88] is an interactive protocol that allows a group of mutually distrusting parties to jointly compute a function over their private inputs without revealing anything beyond the output of the function. Over the years, significant progress has been made towards improving the efficiency of MPC protocols [CGH$^+$18, WJS$^+$19, GS20, GSY21, BGJK21, GPS21, KOS16, HOSS18, CDE$^+$18, GLO$^+$21, DPSZ12] to make them practically viable.

While a wide variety of techniques for efficiency improvements have been developed in different settings based on the corruption threshold, communication model or security guarantee, a common aspect of most modern efficient protocols in all of these settings is that they rely on a *circuit representation* of the function. A limitation of such protocols, however, is that their total communication complexity is at least linear in the size of the circuit. Known techniques for getting sub-linear communication in the circuit size rely on computationally heavy tools such as fully-homomorphic encryption (FHE) [Gen09] or homomorphic secret sharing (HSS) [BGI16]. While there have been recent advancements in improving the efficiency of these methods, they are still far from being practical in many use cases.

As a result, the efficiency of existing efficient protocols is highly dependent on how succinctly a function can be represented using circuits. This is clearly not ideal, since circuits are often not the most efficient way of representing many functions. A common example of such functions are ones that include some kind of *conditional* control flow instructions. When evaluating such functions, a circuit-based MPC will incur communication dependent on the size of the *entire* circuit, while in reality we

only need to evaluate the "active" path (i.e., the path that is actually executed based on the conditional) in the circuit.

It is therefore useful to design efficient MPC protocols for useful classes of functions, where the total communication between the parties only depends on the "active" parts, rather than the entire circuit.

MPC for Conditional Branches. In this work, we focus on one such class of functions, namely, ones that contain conditional branches. As discussed in [HKP21a], a real world example of an application that consists of conditional branches is where a set of servers collectively provide k services and the clients can pay and avail any one of their services (depending on their requirements), without revealing to the servers which service they are availing. Similarly, control flow instructions are also integral to any kind of programming and as observed in [HK21], many kinds of control flow instructions (including repeated and/or nested loops) can be refactored into conditional branches. Such refactorings often result in a *large* number of conditional branches. For such functions, designing MPC protocols where the total communication only depends on the size of the active branch is very useful.

Recently, in a sequence of works [HK20a, HK21], Heath and Kolesnikov made progress in this direction in the two-party setting. They design garbled circuit based *two-party* semi-honest protocols for evaluating functions with conditional branches, where the total communication only depends on the size of the largest branch. In the *multiparty* setting, however, no such protocols are currently known. The recent works of [HKP20, HKP21b] design MPC for conditional branches where they reduce the number of public-key operations required to evaluate conditional branches; however, the total communication in their protocols still depends on the size of all branches. Furthermore, all these protocols only work for Boolean circuits.

Given this state of the art, we consider the following question in this work:

Does there exist an efficient multiparty protocol for securely computing conditional branches, where the total communication only depends on the size of the largest branch?

We remark that all of the above mentioned prior works only focus on the semi-honest setting. The task of designing analogous *maliciously-secure* protocols remains unexplored (both in the two-party and multi-party settings). In this work, we also consider this question.

Our Contributions

We design the first *multiparty* computation protocols for conditional branches, where the communication complexity only depends on the size of the largest branch. Our protocols can support arbitrary number of parties and corruptions. We present both constant and non-constant round variants.

I. Non-Constant Round Branching MPC. Our first contribution is a *semi-honest* MPC for conditional branches, where the communication complexity only depends

on the size of the largest branch. This protocol is capable of computing arithmetic circuits over any field or ring. The round complexity of this protocol depends on the depth of the circuit.

We present this protocol as a generic compiler that can transform a large class of admissible[1] MPC protocols into ones for conditional branches that achieve the aforementioned communication complexity. Several existing concretely efficient protocols including MASCOT [KOS16], SPDZ2k [CDE+18], Overdrive [KPR18], TinyOT [FKOS15] and [HOSS18], [CDN01] can be used with this compiler.

In particular, by instantiating our compiler with a semi-honest admissible (dishonest-majority) MPC protocol with communication complexity $\mathtt{CC}(|C|)$ (where C is the circuit being evaluated), we obtain the following result:

Informal Theorem 3 *Let λ be the security parameter. There exists a semi-honest secure MPC for evaluating conditional branches, that can tolerate arbitrary corruptions and that achieves communication complexity of $O(\mathtt{CC}(|C_{\mathsf{max}}|) + n^2 k\lambda + n^2|C_{\mathsf{max}}|)$, where k is the number of branches in the conditional.*

We also implement this protocol to test its concrete efficiency and compare it to state-of-the-art MPC protocols. More details are provided later in this section.

Extension to Malicious Security. We also present an extension of this protocol to the case of malicious adversaries. Asymptotically, its communication complexity is similar to the semi-honest protocol, except that it incurs a multiplicative overhead dependent on a statistical security parameter.

We view this construction as initial evidence that efficient branching MPC with malicious security is possible. However, we believe that there is significant scope for future improvements towards achieving good concrete efficiency.

II. Constant Round Branching MPC. Our next contribution is a *constant round* MPC for conditional branches, where the communication complexity only depends on the size of the largest branch. This protocol is based on a *multiparty garbling* approach [BMR90] and only supports boolean circuits.

We also present this protocol in the form of a general compiler. Namely, given a MPC protocol with communication complexity $\mathtt{CC}(|C|)$ for evaluating a circuit C, we get the following result:

Informal Theorem 4 *Let λ be the security parameter. There exists a constant-round, semi-honest secure MPC for evaluating conditional branches (represented as Boolean circuits), that can tolerate arbitrary corruptions and that achieves communication complexity of $O(|\mathtt{CC}(\lambda|C_{\mathsf{max}}|) + n^2 k\lambda + n^2\lambda|C_{\mathsf{max}}|)$, where k is the number of branches in the conditional.*

[1] We require the underlying MPC to be such that it evaluates the circuit in a gate-by-gate manner and maintains an invariant that for every intermediate wire in the circuit, the parties collectively hold a sharing of the value induced on that wire during evaluation.

To obtain both of the above results, we adopt a fundamentally different approach as compared to prior works [HK20a, HKP20, HK21, HKP21b] in this area. Specifically, prior works require the parties to locally evaluate *all* the branches. In contrast, in our approach, the parties select the "active" branch and *only execute that branch*. A detailed overview of our approach can be found in the next section.

III. Comparison and Performance Evaluation. To gauge practicality, we implement our non-constant round *semi-honest* compiler and instantiate it using two kinds of protocols:

- *Quadratic Dependence on the Number of Parties:* MP-SPDZ is a common MPC library that contains implementations of the SPDZ protocol [DPSZ12] and its descendants. All of the protocols in this library have total communication with quadratic dependence on the number of parties. We instatiate our compiler with an implementation of MASCOT [KOS16] from this library without modification. Our code is agnostic to which protocol the MPC library is configured; this helps demonstrates that our techniques are generic and block-box. We run benchmarks over simulated LAN and WAN settings. We show that our compiled protocol outperforms naïvely evaluating all the branches in parallel using MASCOT for as few as 8 branches.

- *Linear Dependence on the Number of Parties:* We implement an optimized variant of our compiler that incurs a linear additive overhead in the number of parties, instead of a quadratic overhead. We then test the efficiency of our compiler when instantiated with the CDN protocol [CDN01], which only has a linear dependence on the number of parties. For this, we first implement the CDN protocol. To the best of our knowledge, this is the first known implementation of CDN. Similar to the previous case, we show that our compiled protocol (instantiated using CDN) outperforms naïvely evaluating all the branches in parallel using CDN for 8 branches.

6.2 Technical Overview

Background. All recent works [HK20a, HKP20, HK21, HKP21b] in this area are based on the same principal approach – *the parties evaluate all branches, albeit, only the "active" branch is evaluated on real inputs, while the remaining branches are all evaluated on fake/garbage values*.

For instance, in the two-party setting, [HK20a, HK21], which adopt a garbled circuit based approach, one of the parties garbles all the k branches. It then "stacks" these garblings into a compressed form that is proportional to the length of the largest branch in the circuit. Using some additional information sent by the garbler, the evaluator is able to reconstruct k different garbled circuits, only one of which is a valid garbling of the "active" branch, and the remaining are random strings (or some garbage material). Unaware of the active branch, the evaluator evaluates the k garbled

circuits w.r.t. different branches to obtain k different output labels. These output labels are then filtered with the help of a "multiplexer" to obtain the correct output. Overall, this approach reduces the communication to only depend on the size of the largest branch (the computation complexity, however, is still large).

In the multiparty setting, both [HKP20, HKP21b], follow the same principal approach. These protocols have separate preprocessing and online phases. They require parties to evaluate all branches (including the inactive ones) in the online phase over 0 or some random values and leverage this fact to get savings in the preprocessing phase. As a result, communication in the preprocessing phase only depends on the size of one branch, but the communication in the online phase still depends on the size of all the branches.

Indeed, it is unclear how to extend the stacked garbling approach used in [HK20a, HK21] to get similar savings in communication in the multiparty setting. Recall that the garbler in stacked garbling is required to garble all branches and hence its computation *depends on the size of all branches*. This means that naive approaches that involve distributing the role of the garbler amongst multiple parties are a non-starter as they will incur *communication* proportional to the size of all branches. In order to design a multiparty protocol with similar communication savings as in stacked garbling, we therefore adopt a fundamentally different approach.

Our Approach. In our approach, *the parties select which branch to execute in a "privacy-preserving" manner and only execute that branch.* To facilitate this private selection, both of our constructions (in the non-constant round and constant-round settings) employ a common tool – a variant of oblivious linear evaluation that we refer to as *oblivious inner product* (OIP). In particular, our protocols make use of OIPs with (small) *constant rate*. We show that such OIPs can be easily constructed using low-rate linearly homomorphic encryption schemes, which are known from a variety of assumptions [FV12, CL15, DJ01, PVW08].

In the sequel, we first describe the main ideas underlying our non-constant round constructions. We then proceed to describe our constant-round construction.

Non-Constant Round Branching MPC

We start with the observation that the problem of computing conditional branches bears some similarities to the problem of private function evaluation (PFE) [KM11, MS13, MSS14]. Recall that in PFE, one party has the function and the remaining parties provide inputs. This, in some sense is reminiscent of the problem that we have at hand, albeit with some differences. In particular, in our case, while none of the parties actually knows which function/branch is "active", they all know the set that this branch belongs to. Moreover, the parties collectively hold information about which of these functions to evaluate. This can be viewed as a *distributed variant of PFE*. In light of this observation, we build upon some ideas previously used in the PFE literature.

Private Function Evaluation. In PFE, the function is only known to one of the parties (say party P_1). The security requirements in standard PFE are very similar to that in MPC, with the only additional requirement that the function must remain hidden from all other parties. To achieve this, Mohassel and Sadeghian [MS13] observe that in order to hide a function that is represented in the form of a circuit, there are two components that need to remain hidden – (1) The wire-configuration of the circuit, i.e., how the gates connect with each other, and (2) the function (i.e., addition or multiplication) implemented by each gate in the circuit. They propose a strategy to conceal the above components of a circuit in order to achieve function privacy (without relying on universal circuits). In particular, they start with MPC protocols that work over some kind of secret shares (additive/threshold/authenticated) and evaluate any given circuit in a gate-by-gate manner. These protocols maintain the invariant that for every intermediate wire in the circuit, all parties hold a sharing of the value induced on that wire during evaluation. Many concretely efficient protocols such as [KOS16, HOSS18, CDE$^+$18, GLO$^+$21, DPSZ12], satisfy these requirements. [MS13] propose the following modifications to such MPC protocols to obtain a PFE protocol:

1. *Hiding Wire Configuration:* Each intermediate wire in the circuit has two end points – (1) one is the *source gate*, for which it acts as the outgoing wire and (2) the other is the *destination gate*, for which it acts as the incoming wire. As discussed earlier, for hiding the wire configuration, we need to hide the gate connections, i.e., we want to hide the mapping between the source and destination of each wire in the circuit. For this, [MS13] assign two unique labels to each wire w. One is an outgoing label based on its source gate and second is an incoming label based on whether it acts as left or right input wire to its destination gate. Let π denote the mapping between these incoming and outgoing labels, i.e., let $\pi(i) = j$ denote that a wire that has incoming label i has an outgoing label j. In PFE, this mapping π is only known to the function holding party.

   In order to hide this mapping, [MS13] devise a mechanism to mask the outputs value of each gate and unmask them based on π when this value is used for evaluating the destination gate of this wire. This is executed by sampling an input mask and an output mask for every wire in the circuit. Let $\mathsf{in}_1, \ldots, \mathsf{in}_W$ and $\mathsf{out}_1, \ldots, \mathsf{out}_W$ be the set of these input and output masks, where W is the total number of wires in the circuit. In the preprocessing phase, with the help of the function holding party and the underlying MPC, the parties compute $\Delta_w = \mathsf{in}_w - \mathsf{out}_{\pi(w)}$ for every $w \in [W]$. These Δ_w values are revealed to function holding party in the clear. This processing information helps the parties in using appropriately permuted input and output masks to mask and unmask wire values during evaluation in the online phase. In more detail, the online phase proceeds as follows:

   - Upon evaluating each gate g, the parties use output masks to mask all the outgoing wires of the gate. Let the outgoing wires have labels c and

d respectively, and let u_c and u_d denote these masked outputs. These masked outputs are revealed to all parties in the clear.

- For evaluating a particular gate g, where the two input wires have incoming wire labels a and b, the function holding party computes $A = u_{\pi(a)} + \Delta_a$ and $B = u_{\pi(b)} + \Delta_b$ and sends it to all the parties. The parties subtract their shares of in_a and $\in_b$ from these values to get a sharing of the actual values on which to evaluate gate g.

2. *Hiding Gate Functions:* This is relatively easier. Assume that our arithmetic circuit representation of the function only consists of addition and multiplication gates, let $\mathsf{type}_g = 0$ (and $\mathsf{type}_g = 1$ resp.) denote that gate g is an addition gate (and multiplication gate resp.). For each gate g with incoming wires a and b, we can use the underlying MPC to compute both shares of $a+b$ and $a \cdot b$. The function holding party P_1 can secret share type_g using the underlying MPC and the parties can then choose between shares of $a+b$ and $a \cdot b$ by computing the following using the underlying MPC:

$$(1 - [\mathsf{type}_g])([a+b]) + [\mathsf{type}_g]([a \cdot b]),$$

where we denote $[x]$ as a sharing of a value x using the secret sharing scheme used by the underlying MPC. This allows the parties to evaluate the correct function, without revealing it.

Our Semi-Honest Protocol. In our setting, the parties know the description of all the branches in the conditional and have a secret sharing of the index of the active branch. In order to hide the identity of the active branch, similar to the above approach, we need to hide both the wire configuration and the gate functions of the active branch. We start by listing the barriers in directly adapting the above approach to our setting and then proceed to discuss how we resolve them.

- In the preprocessing phase, computing Δ requires the function holding party to input π to the underlying MPC. In our setting, no party knows the exact value of π.

- In the online phase, A and B values are computed locally by the function holding party in PFE since it already knows the mapping π. This is again a problem in our setting.

- Finally, in order to hide the gate functions in the online phase, the value of each type_g secret shared by the function holding party. But as above, neither party in our setting knows this value.

In order to overcome the above barriers, we crucially rely on the fact that in our setting, while no single party knows the function (or the mapping π), they all know the set that the function belongs to. In other words, given a set of k branches $C_1, \ldots, C_k$, all the parties can locally compute the mappings $\pi_1, \ldots, \pi_k$ corresponding to each branch.

Moreover, the parties also have a secret sharing of the index of the active branch. Let α be the index of the active branch. Our first idea towards resolving the above barriers to is to somehow allow the parties combine their shares of α with $\pi_1, \ldots, \pi_k$ to get a sharing of π_α. However, since the size of $\pi_1, \ldots, \pi_k$ depends on the size of all branches, a naive implementation of this computation will incur communication that depends on the size of $\pi_1, \ldots, \pi_k$.

We get around this by using a new variant of oblivious linear evaluation, which we refer to as oblivious inner product. We now outline our main ideas:

- **Sharing of α:** We work with a unary representation of the index α. In other words, we assume each party have k secret shares, where the α^{th} share is a sharing of 1, while all others are sharings of 0s. Let these shares be denoted by $[b_1], \ldots, [b_k]$

- **Input/Output Masks:** In the preprocessing phase, we use the underlying MPC to sample random input and output masks $\mathsf{in}_1, \ldots, \mathsf{in}_W$ and $\mathsf{out}_1, \ldots, \mathsf{out}_W$, where W is the number of wires in the largest branch. Each party, now locally permutes its shares of input masks based on the k mappings $\pi_1, \ldots, \pi_k$. In more detail, given sharings $[\mathsf{out}_1], \ldots, [\mathsf{out}_W]$, for each $m \in [k]$, the parties locally compute sharings $[\mathsf{out}_{\pi_m(1)}], \ldots, [\mathsf{out}_{\pi_m(W)}]$. Lets denote each $[\mathsf{out}_{\pi_m(1)}], \ldots, [\mathsf{out}_{\pi_m(W)}]$ by $[\overrightarrow{\mathsf{out}_{\pi_m}}]$. If instead of computing shares of π_α, we directly compute re-randomized shares of $[\overrightarrow{\mathsf{out}_{\pi_\alpha}}]$, then the parties can simply compute their shares of Δ_w values as follows

$$\forall w \in [W], [\Delta_w] = [\mathsf{in}_w] - [\mathsf{out}_{\pi_\alpha(w)}]$$

- **Oblivious Inner Product:** For computing re-randomized shares $[\overrightarrow{\mathsf{out}_{\pi_\alpha}}]$, we use a primitive called oblivious inner product (OIP). This is a protocol between two-parties, called the sender and receiver and bears resemblance to oblivious linear evaluation. The sender has inputs $m_0, \ldots, m_k$ and the receiver has inputs $b_1, \ldots, b_k$. At the end of the protocol, the receiver learns $m_0 + \sum_{i \in [k]} b_i m_i$ and the sender learns nothing.

  We use this primitive and a GMW [GMW87] style approach to obtain shares of $\overrightarrow{\mathsf{out}_{\pi_\alpha}}$ as follows: for each pair of parties in the protocol, we run an instance of OIP, where one party acts as the sender and the other acts as the receiver. The inputs of the sender party to this OIP are its shares of $[\overrightarrow{\mathsf{out}_{\pi_1}}], \ldots, [\overrightarrow{\mathsf{out}_{\pi_W}}]$ and a random value X, while the inputs of the receiver are its shares of the unary representation of α. At the end, each party P_i computes its share of $\overrightarrow{\mathsf{out}_{\pi_\alpha}}$ by adding the outputs of each OIP instance where it acted as the receiver and subtracting each X sampled in the OIP instance where it acted as the sender. It is easy to see that these resulting shares are indeed shares of $\overrightarrow{\mathsf{out}_{\pi_\alpha}}$.

  However, note that while the length of the output of each OIP in our case only depends on the size of the largest branch, the length of sender inputs depends on the size of all branches. Therefore, in order to design an MPC protocol where

the overall communication is only proportional to the size of the largest branch, we must use an OIP where the communication only depends on the length of receiver inputs and the output, but is independent of the length of sender inputs. We show that such OIPs can be constructed using linearly homomorphic encryption with constant rate.

- **Online Phase:** Now that we have sharing of Δ_w values that was computed using the mapping π corresponding to the active branch, we can compute shares of the A and B values as follows:

$$[A] = \sum_{m \in [k]} [b_1] u_{\pi_1(a)} + [\Delta_a] \quad \text{and} \quad [B] = \sum_{m \in [k]} [b_1] u_{\pi_1(b)} + [\Delta_b]$$

We note that most linearly homomorphic secret sharing schemes allow such computations to be done non-interactively and hence it does not incur any overhead in the communication complexity. Shares of type_g for every gate g can also be computed in a similar manner.

Extension to Malicious Security. While the basic outline of our protocol remains the same even in the malicious setting, we need to do a little more work to make the above protocol secure against a malicious adversary. In particular, we need to ensure that the inputs used by the parties in the OIP instances are consistent with values/shares computed by them using the underlying MPC. For this we propose to add the following consistency checks:

Receiver's Input Consistency. We start by using an OIP that is secure against a malicious reciever. In order to ensure that receiver uses valid sharings of the active branch, we implement a kind of MAC check using the underlying MPC. In particular, in the OIP execution, the sender samples $k+1$ random values and appends them to its inputs. Now when the receiver computes the output of the OIP, it also learns an inner product of these random values with its shares of the active branch (we refer to this as the MAC value for this OIP). We now use the underlying MPC to compute the exact same value. In particular, the sender sends the $k+1$ random values that it sampled in the OIP as input to the underlying MPC, while the receiver sends the MAC value learnt from the output of the OIP. We allow the underlying MPC to now check if the MAC value indeed corresponds to an inner product of the receivers shares of the active branch and the random values input by the sender. We note that since the length of the receiver's input is independent of the size of all branches, computing this MAC value inside the MPC does not incur too much overhead.

Sender's Input Consistency. Recall that the inputs of the sender to the OIP depend on the size of all branches, and hence we cannot hope to use the kind of check that we used for ensuring receiver consistency. Moreover, since the length of the sender message is much shorter than the length of its inputs, we also cannot hope to use an OIP with malicious sender security that can somehow extract the sender's inputs.

Therefore, instead we continue to work with an OIP that is secure against a semi-honest sender but augment it with a cut-and-choose style approach. In particular, we sample multiple copies of the masks and compute delta values using OIPs for each of those copies. We also ask the sender to commit (using compressive commitments) to the inputs and randomness used for computing each of its sender messages. At the end of all OIP instantiations, we use the underlying MPC to sample a random subset and reveal the shares of masks of all parties for that subset. The senders also send the randomness used by them in the sender messages of this opened subset. Given this information, the parties can verify if the senders behaved honestly and used consistent shares in the opened instances. We use the remaining unopened instances to run multiple copies of the online phase and take a majority to decide the final output. Due to the use of cut-and-choose, the communication complexity of our maliciously secure protocol is proportional to $\delta \times$ the cost of computing the largest branch. Nevertheless, as discussed in the introduction, this is still useful for conditionals with large number of branches.

Constant Round (Semi-Honest) Protocol

Beaver, Micali, and Rogaway (BMR) [BMR90] proposed a general template for constructing *constant round* MPC from existing generic *non-constant round* MPC. The main observations underlying their technique were – (1) round complexity of more generic non-constant round protocols depends on the depth of the function being computed and (2) garbling [Yao86] a functionality/circuit is a constant depth procedure.

The parties can leverage these observations to first execute a *garbling phase*, where they compute a garbled circuit of the function (that they wished to evaluate) using the non-constant round protocol. This phase will require a constant number of rounds. Given this garbled circuit, they then proceed to the *evaluation phase*, where each party locally evaluates the garbled circuit to learn the output. This phase requires no interaction and hence the overall protocol runs in a constant number of rounds.

More concretely, in the garbling phase, the parties collectively sample two keys $\mathsf{k}_{w,0}, \mathsf{k}_{w,1}$ for every wire w in the circuit. The garbled table for each gate g in the circuit with incoming wires a, b and outgoing wire c, consists of the following four rows, corresponding to $\alpha, \beta \in \{0,1\}$:

$$ct_{\alpha,\beta} = \mathsf{PRF}_{\mathsf{k}_{a,\alpha}}(g) + \mathsf{PRF}_{\mathsf{k}_{b,\beta}}(g) + \mathsf{k}_{c,g(\alpha,\beta)}$$

Branching MPC using BMR Template. The generality of the BMR approach immediately makes it compatible with our non-constant round semi-honest protocol (from Section 6.2). Indeed, in the garbling phase, parties can use that protocol to compute a garbled circuit for the active branch. During the evaluation phase, however, since the parties do not know which branch the garbled circuit corresponds to, they can evaluate it for every branch and obtain the corresponding output wire labels. Note that only the labels obtained by evaluating w.r.t. to the active branch actually

correspond to a *valid set of abels*. Finally, via interaction, parties can determine the output corresponding to the "valid" set of output labels. The complexity of this last step is independent of the circuit size and only depends on the number of branches times the output length.

While this yields a simple baseline constant round MPC for conditional branches, it is highly inefficient. Since no party knows the keys $\mathsf{k}_{a,\alpha}, \mathsf{k}_{b,\beta}$ in their entirety, they must evaluate the PRF (on these keys) inside an MPC protocol. Since, the circuit representations of PRF's are typically massive, this protocol is unlikely to be concretely efficient. As such, for concrete efficiency, we require a protocol that only makes a *black-box* use of cryptography.

Towards Black-Box use of Cryptography. Damgård and Ishai [DI05] proposed a variant of the above BMR template that enables parties to evaluate the PRF outside the MPC, thereby only making a black-box use of cryptography.

Specifically, in their approach, each party P_i samples two keys $\mathsf{k}^i_{w,0}, \mathsf{k}^i_{w,1}$ for every wire w in the circuit. In other words, the cumulative keys associated with every wire is a concatenation of all the parties' keys. The garbled table for each gate g in the circuit with incoming wires a, b and outgoing wire c, consists of the following $4 \cdot n$ rows, corresponding to $\alpha, \beta \in \{0,1\}$ and $i \in [n]$:

$$ct^i_{\alpha,\beta} = \bigoplus_{m=1}^{n} \mathsf{PRF}_{\mathsf{k}^m_{a,\alpha}}(g\|i) \; + \; \bigoplus_{m=1}^{n} \mathsf{PRF}_{\mathsf{k}^m_{b,\beta}}(g\|i) + \mathsf{k}^i_{c,g(\alpha,\beta)}$$

It is easy to see that unlike the BMR approach, here the parties are only required to evaluate the PRF on their own keys, which can be done locally and the resulting PRF evaluation can be fed as input to the underlying MPC implementing the garbling functionality.

In our setting, however, this approach posits a fundamental barrier. Recall that for evaluating conditional branches, we want to garble the active branch without revealing the index of the active branch. For this, while garbling any gate (say the j^{th} gate), it is imperative that the parties remain oblivious to both the functionality associated with it and its incoming and outgoing wires. As a result, the parties are unaware of which keys $\mathsf{k}^i_{a,\alpha}, \mathsf{k}^i_{b,\beta}$ to use for computing the corresponding ciphertexts, and hence cannot evaluate the PRF on those keys *locally*. A natural approach to overcome this problem is to perform this evaluation within an MPC; however, we are then back to the realm of non-black-box use of cryptography. As such it is unclear how to directly adapt this approach to our setting, while making a black-box use of cryptography.

Garbling using Key-Homomorphic PRFs. To overcome the above barrier, we explore the work of Ben-Efraim et al. [BLO17] who presented an alternative template for multiparty garbling, using *key-homomorphic* PRFs. These are PRFs with the following property: $\mathsf{PRF}_{\mathsf{k}_1}(m) \tilde{+} \mathsf{PRF}_{\mathsf{k}_2}(m) = \mathsf{PRF}_{\mathsf{k}_1 \tilde{\cdot} \mathsf{k}_2}(m)$, where $\tilde{+}$ and $\tilde{\cdot}$ are some operations. As before, each party samples two keys for every wire in the circuit and given such a PRF, the parties the compute each ciphertext as follows:

$$ct_{\alpha,\beta} = \widetilde{\sum}_{m\in[n]} \left(\mathsf{PRF}_{\mathsf{k}^m_{a,\alpha}}(g) \tilde{+} \mathsf{PRF}_{\mathsf{k}^m_{b,\beta}}(g)\right) \tilde{+} \left(\widetilde{\prod}_{m\in[n]} \mathsf{k}^m_{c,g(\alpha,\beta)}\right)$$

It is easy to see that similar to the previous approach, each party here is only required to evaluate the PRF on its own key, which can be done locally. At first, it might seem that in our setting, the same problem (as before) still persists. Indeed, for local PRF evaluation, the parties are required to know which key to use, which as discussed earlier is not possible when the parties are required to obliviously garble one of the conditional branches. However, we observe that homomorphism of the PRF can be leveraged here to resolve this problem.

Lets assume that the parties start by ordering the gates and wires in every branch in some canonical order. Now, when garbling the j^{th} gate of the active branch, they must choose the appropriate keys from all the keys associated with the j^{th} gate in every branch. We also assume that the parties have a sharing of the unary representation of the index associated with the active branch. The parties can now use multiple instances of OIP (as in our non-constant round protocols) to obtain shares of the keys associated with the two incoming wires of the j^{th} gate in the active branch.

Consider a key homomorphic PRF where both $\tilde{+}$ and $\tilde{\cdot}$ are the same operation associated with the reconstruction algorithm of the secret sharing scheme used in the undelying MPC, i.e., $[\mathsf{PRF}_{\mathsf{k}}(m)] = \mathsf{PRF}_{[\mathsf{k}]}(m)$. This PRF can now be used along with the above observation to compute a garbling of the active branch as follows: for simplicity let's assume that each branch is of the same size and has W wires. The parties start by collectively sampling $2W$ keys. For garbling the j^{th} gate, for each $\alpha,\beta \in \{0,1\}$, they use OIPs to compute shares $[\mathsf{k}_{a,\alpha}], [\mathsf{k}_{b,\beta}]$ and $[\mathsf{k}_{c,g(\alpha,\beta)}]$, where a,b are the incoming and c is the outgoing wire of the j^{th} gate in the active branch and g is the function computed by this gate. Parties can now locally evaluate the PRF on these shares and use the underlying MPC to compute shares of the ciphertexts as follows:

$$[ct_{\alpha,\beta}] = \mathsf{PRF}_{[\mathsf{k}_{a,\alpha}]}(j) + \mathsf{PRF}_{[\mathsf{k}_{b,\beta}]}(j) + [\mathsf{k}_{c,g(\alpha,\beta)}]$$

Upon computing this garbled circuit for the active branch, similar to the baseline solution, parties evaluate it w.r.t. all the branches and then run a "mini-MPC" to filter out the valid labels and determine the final output.

Instantiating Key Homomorphic PRF. Most existing dishonest majority MPC protocols [KOS16, HOSS18, CDE$^+$18, GLO$^+$21, DPSZ12] use additive secret sharing. To use the above ideas with such protocols, we need an additively key-homomorphic PRF, i.e., where $\mathsf{PRF}_{\mathsf{k}_1}(m) + \mathsf{PRF}_{\mathsf{k}_2}(m) = \mathsf{PRF}_{\mathsf{k}_1+\mathsf{k}_2}(m)$. Unfortunately, key homomorphic PRFs are currently only known from the DDH assumption [NPR99, BLMR13] and those PRFs do not achieve a similar additive homomorphism.

Ben-Efraim et al. [BLO17] observed that instead of a PRF, it suffices to use a (decisional) ring LWE based random function here. This function is of the form: $F = f_k : \mathscr{R}_p \to \mathscr{R}_p | f_k(a) = a \cdot k + e$, where $p = 2N+1$ is a prime, N is a power of two, $\mathscr{R}_p = \mathbb{Z}_p[X]/(X^N+1)$ and a,k, and e are polynomials in the ring and the coefficients of e come from a gaussian distribuition. Since a is public, it is easy to see that given

additive shares of the key k and error e, it is possible for the parties to locally compute shares of the above function. As is standard when using LWE/RLWE, encrypting using such a random function typically requires multiplying the message (before adding it to the output of this function) with the size of the range from which the message comes from. In the case of garbling, since both the message and keys come from the same distribution, as shown in [BLO17], this requires choosing the parameters carefully and additionally requires sampling the keys from a gaussian distribution. However, since the parties only need to compute additive shares of these keys and errors, this can be done easily by requiring the parties to sample their shares from appropriate distributions. We defer more details to the technical sections.

6.3 Preliminaries

Secure Multiparty Computation

A secure multi-party computation protocol (MPC) is a protocol executed by n parties $\mathscr{P} = \{P_1, \cdots, P_n\}$ for a functionality $\mathscr{F}$. We allow for parties to exchange messages simultaneously. In every round, every party is allowed to exchange messages with other parties using different communication channels, depending on the model. A protocol is said to have k *rounds* if it proceeds in k distinct and interactive rounds.

Adversarial Behavior

One of the primary goals in MPC is to protect the honest parties against dishonest behavior of the corrupted parties. This is usually modeled using a central adversarial entity, that controls the set of corrupted parties and instructs them on how to operate. That is, the adversary obtains the views of the corrupted parties, consisting of their inputs, random tapes and incoming messages, and provides them with the messages that they are to send in the execution of the protocol.

In this work we consider two types of adversaries. A *semi-honest adversary* is "honest but curious" where it always follows the instructions of the protocol but might try to learn extra information by analyzing the transcript of the protocol later. On the other hand, *a malicious adversary* can deviate from the protocol and instruct the corrupted parties to follow any arbitrary strategy.

We provide the basic definitions for secure multiparty computation according to the real/ideal paradigm [Gol04]. Informally, a protocol is considered secure if whatever an adversary can do in the real execution of protocol, can be done also in an ideal computation, in which an uncorrupted trusted party assists the computation.

Security Definitions

Real World. The real world execution of a protocol $\Pi = (P_1, \ldots, P_n)$ begins by an adversary $\mathscr{A}$ selecting any arbitrary subset of parties $\mathscr{I}$ to corrupt. The parties then engage in an execution of a real n-party protocol Π. Throughout the execution of

Π, the adversary $\mathscr{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of Π. At the conclusion of the protocol, each honest party outputs all the outputs it obtained in the computations. Malicious parties may output an arbitrary PPT function of the view of $\mathscr{A}$. This joint execution of Π under $(\mathscr{A}, \mathscr{I})$ in the real model, on input vector $\overrightarrow{x} = (x_1, \ldots, x_n)$, auxiliary input z and security parameter λ, denoted by $\mathsf{REAL}_{\Pi,\mathscr{I},\mathscr{A}(z)}\left(1^\lambda, \overrightarrow{x}\right)$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathscr{A}(z)$ resulting from this protocol interaction.

Ideal World. We now present standard definitions of ideal-model computations.

An ideal computation of an n-party functionality $\mathscr{F}$ on input $\overrightarrow{x} = (x_1, \ldots, x_n)$ for parties $(P_1, \ldots, P_n)$ in the presence of an ideal-model adversary $\mathscr{A}$ controlling the parties indexed by $\mathscr{I} \subset [n]$, proceeds via the following steps.

Sending inputs to trusted party: For each $i \notin \mathscr{I}$, P_i sends its input x_i to the trusted party. If $i \in \mathscr{I}$, the adversary may send to the trusted party any arbitrary input for the corrupted party P_i. Let x_i' be the value actually sent as the i^{th} party's input.

Early abort: The adversary $\mathscr{A}$ can abort the computation by sending an abort message to the trusted party. In case of such an abort, the trusted party sends $\perp$ to all parties and halts.

Trusted party answers adversary: The trusted party computes $(y_1, \ldots, y_n) = \mathscr{F}(x_1', \ldots, x_n')$ and sends y_i to party P_i for every $i \in \mathscr{I}$.

Late abort: The adversary $\mathscr{A}$ can abort the computation (after seeing the outputs of corrupted parties) by sending an abort message to the trusted party. In case of such abort, the trusted party sends $\perp$ to all honest parties and halts. Otherwise, the adversary sends a continue message to the trusted party.

Trusted party answers remaining parties: The trusted party sends y_i to P_i for every $i \notin \mathscr{I}$.

Outputs: Honest parties always output the message received from the trusted party and the corrupted parties output nothing. The adversary $\mathscr{A}$ outputs an arbitrary function of the initial inputs x_i s.t. $i \in \mathscr{I}$, the messages received by the corrupted parties from the trusted party and its auxiliary input.

Security Having defined the real and ideal models, we can now define security of protocols according to the real/ideal paradigm.

Definition 24 *Let $\mathscr{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party functionality and let Π be a probabilistic polynomial-time protocol computing $\mathscr{F}$. The protocol Π t-securely computes $\mathscr{F}$, if for every probabilistic polynomial-time real-model adversary $\mathscr{A}$,*

there exists a probabilistic polynomial-time simulator $\mathscr{S}$ for the ideal model, such that for every $\mathscr{I} \subset [n]$ of size at most t, it holds that

$$\left\{\mathsf{REAL}_{\Pi,\mathscr{I},\mathscr{A}(z)}\left(1^\lambda, \overrightarrow{x}\right)\right\}_{(\overrightarrow{x},z)\in(\{0,1\}^*)^{n+1},\lambda\in\mathbb{N}} \approx_c \left\{\mathsf{IDEAL}_{\mathscr{F},\mathscr{I},\mathscr{S}(z)}(1^\lambda, \overrightarrow{x})\right\}_{(\overrightarrow{x},z)\in(\{0,1\}^*)^{n+1},\lambda\in\mathbb{N}}$$

6.4 Oblivious Inner Product

In this section, we define a variant of oblivious linear evaluation (OLE), which we refer to as *oblivious inner product* (OIP). OIP is a protocol between two parties, called the *sender* and *receiver* respectively. The sender has inputs $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k})$ in some domain (say $\mathbb{dom}^m$), and receiver has inputs $(b_1, \ldots, b_k)$ in the same domain $\mathbb{dom}$. At the end of the protocol, the receiver should learn $\overrightarrow{m_0} + \sum_{i\in[k]} b_i \overrightarrow{m_i}$ and nothing more, while the sender should learn nothing about the reciever inputs $b_1, \ldots, b_k$.

For our constructions, we consider two variants of OIP, a *semi-honest* version and one that is secure against a *malicious receiver*. We now define the syntax and the security guarantees of a two-message OIP protocol in the plain model. The definitions can be naturally extended to the CRS model.

Definition 25 (Two-Message Oblivious Inner Product) *A two-message oblivious inner product between a receiver* R *and a sender* S *is defined by a tuple of 3 PPT algorithms* $(\mathsf{OIP}_\mathsf{R}, \mathsf{OIP}_\mathsf{S}, \mathsf{OIP}_\mathsf{out})$. *Let* λ *be the security parameter. The receiver computes* $\mathsf{msg}_\mathsf{R}, \rho$ *as the evaluation of* $\mathsf{OIP}_\mathsf{R}(1^\lambda, (b_1, \ldots, b_k))$, *where* $(b_1, \ldots, b_k) \in \mathbb{dom}^k$ *is the receiver's input. The receiver sends* msg_R *to the sender. The sender then computes* msg_S *as the evaluation of* $\mathsf{OIP}_\mathsf{S}(1^\lambda, \mathsf{msg}_\mathsf{R}, (\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}))$, *where* $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) \in \mathbb{dom}^{m\times(k+1)}$ *are sender's inputs. The sender sends* msg_S *to the receiver. Finally, the receiver computes the output by evaluating* $\mathsf{OIP}_\mathsf{out}(\rho, \mathsf{msg}_\mathsf{R}, \mathsf{msg}_\mathsf{S})$.

A semi-honest OIP satisfies correctness, security against semi-honest receiver and semi-honest sender*, while the malicious variant satisfies* correctness, security against semi-honest sender and malicious receiver*, which are defined as follows:*

- ***Correctness**: For each* $(\overrightarrow{m_0}, \ldots, \overrightarrow{m_k}) \in \mathbb{dom}^{m\times(k+1)}$ *and* $(b_1, \ldots, b_k) \in \mathbb{dom}^k$, *the following holds*

$$\Pr\left[\begin{array}{c}(\rho, \mathsf{msg}_\mathsf{R}) \leftarrow \mathsf{OIP}_\mathsf{R}\left(1^\lambda, (b_1, \ldots, b_k)\right) \\ \mathsf{msg}_\mathsf{S} \leftarrow \mathsf{OIP}_\mathsf{S}\left(1^\lambda, \mathsf{msg}_\mathsf{R}, (\overrightarrow{m_0}, \ldots, \overrightarrow{m_k})\right)\end{array} : \mathsf{OIP}_\mathsf{out}(\rho, \mathsf{msg}_\mathsf{R}, \mathsf{msg}_\mathsf{S}) = \overrightarrow{m_0} + \textstyle\sum_{i\in[k]} b_i \overrightarrow{m_i}\right] = 1$$

- ***Security Against Semi-Honest Sender**: The following holds for any* $(b_1, \ldots, b_k) \in \mathbb{dom}^k$ *and* $(b'_1, \ldots, b'_k) \in \mathbb{dom}^k$, *where* $\exists i \in [k]$ *s.t.* $b_i \neq b'_i$

$$\left\{(\mathsf{msg}_\mathsf{R}, \rho) \leftarrow \mathsf{OIP}_\mathsf{R}\left(1^\lambda, (b_1, \ldots, b_k)\right) : \mathsf{msg}_\mathsf{R}\right\} \approx_c \left\{(\mathsf{msg}'_\mathsf{R}, \rho') \leftarrow \mathsf{OIP}_\mathsf{R}\left(1^\lambda, (b'_1, \ldots, b'_k)\right) : \mathsf{msg}'_\mathsf{R}\right\}.$$

- ***Security Against Semi-Honest Receiver****: For every PPT adversary $\mathscr{A}$ corrupting the receiver, there exists a PPT simulator $\mathscr{S}_{\mathsf{R}}$ such that for any choice of $(b_1,\ldots,b_k) \in \mathbb{dom}^k$ and $(\overrightarrow{m_0},\ldots,\overrightarrow{m_k}) \in \mathbb{dom}^{m\times(k+1)}$, the following holds:*

$$\mathsf{OIP}_{\mathsf{S}}\left(1^{\lambda},\mathsf{msg}_{\mathsf{R}},(\overrightarrow{m_0},\ldots,\overrightarrow{m_k})\right) \approx_c \mathscr{S}_{\mathsf{R}}(1^{\lambda},\rho,\mathsf{msg}_{\mathsf{R}},\overrightarrow{m_0}+\sum_{i\in[k]} b_i\overrightarrow{m_i}),$$

  where $(\mathsf{msg}_{\mathsf{R}},\rho) \leftarrow \mathsf{OIP}_{\mathsf{R}}(1^{\lambda},(b_1,\ldots,b_k))$.

- ***Security against a Malicious Receiver****: For every PPT adversary $\mathscr{A}$ corrupting the receiver, there exists a PPT simulator $\mathscr{S}_{\mathsf{R}} = (\mathscr{S}^1_{\mathsf{R}},\mathscr{S}^2_{\mathsf{R}})$, such that for any choice of $(\overrightarrow{m_0},\ldots,\overrightarrow{m_k}) \in \mathbb{dom}^{m\times(k+1)}$, the following holds:*

$$\left|\Pr\left[\mathsf{IDEAL}_{\mathscr{S}_{\mathsf{R}},\mathscr{F}_{\mathsf{OIP}}}(1^{\lambda},\overrightarrow{m_0},\ldots,\overrightarrow{m_k})=1\right]-\Pr\left[\mathsf{REAL}_{\mathscr{A},\mathsf{OIP}}(1^{\lambda},\overrightarrow{m_0},\ldots,\overrightarrow{m_k})=1\right]\right| \leq \tfrac{1}{2}+\mathsf{negl}(\lambda).$$

  Where experiments $\mathsf{IDEAL}_{\mathscr{S}_{\mathsf{R}},\mathscr{F}_{\mathsf{OIP}}}$ *and* $\mathsf{REAL}_{\mathscr{A},\mathsf{OIP}}$ *are defined as follows:*

Exp $\mathsf{IDEAL}_{\mathscr{S}_{\mathsf{R}},\mathscr{F}_{\mathsf{OIP}}}\left(1^{\lambda},\overrightarrow{m_0},\ldots,\overrightarrow{m_k}\right)$:

- $\mathsf{msg}_{\mathsf{R}} \leftarrow \mathscr{A}(1^{\lambda})$
- $(b_1,\ldots,b_k) \leftarrow \mathscr{S}^1_{\mathsf{R}}(1^{\lambda},\mathsf{msg}_{\mathsf{R}})$
- $\mathsf{out} \leftarrow \mathscr{F}_{\mathsf{OIP}}(\overrightarrow{m_0},\ldots,\overrightarrow{m_k},b_1,\ldots,b_k)$
- $\mathsf{msg}_{\mathsf{S}} \leftarrow \mathscr{S}^2_{\mathsf{R}}(1^{\lambda},\mathsf{out},\mathsf{msg}_{\mathsf{R}})$
- Output $\mathscr{A}(\mathsf{msg}_{\mathsf{S}})$

Exp $\mathsf{REAL}_{\mathscr{S}_{\mathsf{R}},\mathscr{F}_{\mathsf{OIP}}}\left(1^{\lambda},\overrightarrow{m_0},\ldots,\overrightarrow{m_k}\right)$:

- $\mathsf{msg}_{\mathsf{R}} \leftarrow \mathscr{A}(1^{\lambda})$
- $\mathsf{msg}_{\mathsf{S}} \leftarrow \mathsf{OIP}_{\mathsf{S}}\left(1^{\lambda},\mathsf{msg}_{\mathsf{R}},(\overrightarrow{m_0},\ldots,\overrightarrow{m_k})\right)$
- Output $\mathscr{A}(\mathsf{msg}_{\mathsf{S}})$

We present a construction of such OIPs from constant rate linearly homomorphic encryption in Section 6.9.

6.5 MPC Interface

As discussed in the introduction, all of our compilers make use of an underlying secure computation protocol with certain properties. In this section, we describe the properties that we want from these underlying protocols.

We model these requirements as a reactive functionality (denoted as $\mathscr{F}_{\mathsf{mpc}}$). At a high level, we require secret sharing based MPC that evaluate a given circuit in a gate-by-gate manner and maintain an invariant that the parties hold a secret sharing of the values induced on each intermediate wire in the circuit. A formal description of this reactive functionality appears in Figure 6.1.

For ease of notation, in our protocol descriptions, we shall let $[varid]$ denote the value stores by the functionality under $(varid,a)$; and we will write $[z]=[x]+[y]$ as a shorthand for calling **Add** and $[z]=[x]\cdot[y]$ as a shorthand for calling **Multiply**.

And by abuse of notation, we will let *varid* denote the value, x, of the data item held in location $(varid, x)$. We use $[x]_i$ to denote the share of x given to party P_i in the underlying MPC.

To the best of our knowledge, most secret sharing based protocols [KOS16, HOSS18, CDE+18, DPSZ12, CDN01] securely implement this reactive functionality in the presence of a malicious adversary who can corrupt arbitrary number of parties. Moreover, most of these protocols are capable of evaluating circuits over any field/ring.

It is easy to see that any such secret sharing based MPC that evaluates the circuit in a gate-by-gate manner and maintains the invariant that parties hold shares of all intermediate wires in the circuit will trivially have support for the **Initialize Input, Initialize constant, Add, Add by const, Multiply, Multiply by const, Function** and **Output Private Shares** calls. Moreover, since the multiplication in these protocols typically requires parties to actually generate and compute shares of random values, the **Random** call is also implemented by these protocols. We now discuss how the remaining calls can be implemented in both the semi-honest and malicious settings.

Semi-Honest Setting. The only other calls used in our semi-honest protocols are **Random Bit** and **Output**. As observed in some of these protocols, **Random Bit** is also very easy to implement (especially in the semi-honest setting). This is done by requiring each party P_i to randomly sample $b_i \in \{1,-1\}$ and secret share it amongst all the parties. The parties then add all the shares obtained from all parties (let the resulting shares be $[s]$) and then compute $\frac{[s]+1}{2}$. The resulting shares will be of a random bit. **Share Zero** can be realized with semi-honest security by having every party secret share 0 and then requiring each party to locally sum up its shares. Finally, it is easy to see that the **Output** call can also be easily implemented, since the parties actually hold shares of all intermediate values. To reconstruct the output, they can simply broadcast their respective shares to all parties and then run the reconstruction algorithm.

Malicious Setting. While protocols such as SPDZ [DPSZ12] and its descendants [KOS16, HOSS18, CDE+18] (that use MACs w.r.t. a global key) delegate the check that ensures that these shares are indeed consistent with the "correct" values to the end of the protocol, we show that these protocols still securely implement all remaining calls in the $\mathscr{F}_{\mathsf{mpc}}$ functionality.

Intuitively, since these protocols delegate the malicious security/consistency checks to the end the protocol, the only place where we need to ensure that the shares held by the parties for any particular wire are indeed consistent and correct is when those values are reconstructed or are used outside of this MPC protocol, i.e., in the OIP and when the outcome of OIP is returned to the MPC. The subcalls inside $\mathscr{F}_{\mathsf{mpc}}$ that are really affected by this are **Initialize Input, Random, Share Zero, Check Zero** and **Output Shares** and **Output**. As discussed above, **Initialize Input** and **Random** are already implemented by these protocols.

- **Check Zero:** For this sub-call, we observe that given authenticated additive shares $([x_1],[m_1])$, $([x_2],[m_2])$, with $m_1 = k*x_1$, $m_2 = k*x_2$ where k is the global

MAC key, parties can compute $[m] = [m_1] - [m_2]$ locally, followed by having each player P_i first commit and then broadcast its share $[m]_i$ to reconstruct $[m]$ and check if $m = \sum_i m_i = 0$.

- **Share Zero:** For this we can augment the semi-honest **Share Zero** protocol described above with an asymptotically efficient batch-wise check to ensure malicious security. Specifically, to verify the outputs of the ℓ semi-honest **Share Zero** calls $[x_1], \ldots, [x_\ell]$, parties can publicly sample ℓ random values $\{r_i\}_{i=1}^{\ell}$ and compute a random linear combination $[r] = \sum_{i=1}^{\ell} r_i[x_i]$ followed by running the **Check Zero** call on $[r]$ and a trivial sharing of 0 (each party P_i's share is 0).

- **Output and Output Share:** As discussed above authenticated shares in the above protocols are of the form $([x], [m])$, where $m = k * x$ and k is the global MAC key. For both of these sub-calls, the parties first broadcast their shares $[x]$ and reconstruct. Then the parties can compute $x \cdot [k]$ and run **Check Zero** to check if the resulting shares reconstruct to the same value as the shares $[m]$. This is very similar to "MAC check" subprotocol already implemented in [KOS16].

  We note that the above proposed protocols only reveal shares $[x]$ and not $[m]$. Indeed, revealing all shares of both x an m will trivially give away the global MAC key and make the protocol insecure. To make this compatible with our maliciously secure protocol, we assume that when the parties use the shares generated via $\mathscr{F}_{\mathsf{mpc}}$ outside of $\mathscr{F}_{\mathsf{mpc}}$ (i.e., to compute the OIP messages), they can do so on the "unauthenticated shares", i.e., on only the $[x]$ part and not on the $[m]$ part. Now, before, using the shares obtained as output of this OIP in $\mathscr{F}_{\mathsf{mpc}}$, we can make them "authenticated" by computing the corresponding $[m]$ shares for this output. This can be done trivially, since the parties hold a secret sharing of the global MAC key. This is a standard approach used in many of the above protocols including MASCOT [KOS16].

  Moreover, we remark that the above proposed modification does not cause our compiler or the compiled protocols to be insecure in any way. This is because, the authentication mechanism used on the shares is only specific to $\mathscr{F}_{\mathsf{mpc}}$ and not to the primitives used outside of it. As a result, outside of $\mathscr{F}_{\mathsf{mpc}}$, an adversary can easily modify the authenticated shares in whatever way they want. Hence, in principle the following strategies are equivalent – (1) where the computations done outside of $\mathscr{F}_{\mathsf{mpc}}$ are performed on authenticated shares. (2) where the computations done outside of $\mathscr{F}_{\mathsf{mpc}}$ are performed on unauthenticated shares, but we authenticate the output of those computations before they are used in $\mathscr{F}_{\mathsf{mpc}}$ again.

6.6 Non-Constant Round Semi-Honest Branching MPC

In this section, we present our semi-honest compiler for distributed computation of a circuit with conditional branches.

Functionality $\mathscr{F}_{\mathsf{mpc}}$

Initialize Input: On input $(initinp, varid, P_i)$ from P_i (for each $i \in [n]$) with a fresh identifier *varid* the functionality stores $(varid, [x])$.
Initialize constant: On input $(initconst, constid, c)$ from each P_i ($i \in [n]$) with a fresh identifier *varid* the functionality stores $(const, c)$.
Random: On command $(rand, varid)$ from all parties, with a fresh identifier *varid*, the functionality selects a random value r, stores $(varid, [r])$ and sends the respective share $[r]_i$ to party P_i (for each $i \in [n]$)
Random Bit: On command $(bitrand, varid)$ from all parties, with a fresh identifier *varid*, the functionality selects a random bit $b \in \{0,1\}$, stores $(varid, [b])$ and sends the respective share $[b]_i$ to party P_i (for each $i \in [n]$)
ShareZero: On command $(sharezero, varid)$ from all parties, with a fresh identifier *varid*, the functionality computes, stores $(varid, [0])$ and sends the respective share $[0]_i$ to party P_i (for each $i \in [n]$).
Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, [x]), (varid_2, [y])$ and stores $(varid_3, [x+y])$.
Add by const: On command $(add, constid_1, varid_2, varid_3)$ from all parties (if $constid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(constid_1, c), (varid_2, [x])$ and stores $(varid_3, [c+x])$.
Multiply: On input $(mult, varid_1, varid_2, varid_3)$ from all parties (if $varid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, [x]), (varid_2, [y])$ and stores $(varid_3, [x \cdot y])$.
Multiply by const: On command $(mult, constid_1, varid_2, varid_3)$ from all parties (if $constid_1$, $varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(constid_1, c), (varid_2, [x])$ and stores $(varid_3, [c \cdot x])$.
Function: On input $(func, f, varid_1, \ldots, varid_n, varid_{\mathsf{out}})$ from all parties, the functionality retrieves $(varid_1, [x_1]), \ldots, (varid_n, [x_n])$ and stores $(varid_{\mathsf{out}}, [f(x_1, \ldots, x_n)])$.
Output Shares: On input $(outshare, varid)$ from all parties, the functionality retrieves $(varid, [x])$ and outputs all shares $[x]$ to all parties.
Output Private Shares: On input $(outprivshare, varid)$ from all parties, the functionality retrieves $(varid, [x])$ and outputs the respective share $[x]_i$ to party P_i (for each $i \in [n]$).
Check Zero: On input $(fcheckzero, varid_1, varid_2)$ from all parties, the functionality retrieves $(varid_1, [x_1]), (varid_2, [x_2])$ and outputs 1 w.h.p if $x_1 = x_2$ and otherwise it outputs 0 and aborts.
Output: On input $(out, varid)$ from all honest parties (if *varid* is present in memory), the functionality retrieves $(varid, [x])$ and outputs x to all players.

Figure 6.1: A Required Ideal Functionality for MPC

Let the circuit/function be such that it consists of an initial sub-function f_1, followed by the k branches and then a sub-function f_2. We assume that the parties have access to $\mathscr{F}_{\mathsf{mpc}}$ (see Figure 6.1). When evaluated using $\mathscr{F}_{\mathsf{mpc}}$, the output of f_1 is a secret sharing of the inputs to the branching part and a secret sharing of the

unary representation of the index associated with the branch that needs to be executed (henceforth referred to as the active branch). The output of the branching part is a secret sharing of the inputs to the function f_2.

Given a circuit C, we assume that the parties decide on some canonical ordering of the gates in the circuit, such that gate i only takes as inputs the values output by the gates $j < i$. We assume w.l.o.g. that the i^{th} gate in C has fan-in 2 and the outgoing wire of any gate can act as the incoming wire for any number of gates.[2]

For simplicity, we assume that all branches are of the same size and have G gates. Our protocol can be easily extended to the scenario where the branches are of varying sizes by suitably padding the smaller branches with fake gates. Let ℓ be the length of inputs to the branching part of the function. For evaluating this part, we assume that there are ℓ input gates that are common to all branches. We set both the incoming and outgoing labels for the wires coming out of these gates as $1,\ldots,\ell$ respectively. For each branch $m \in [k]$, and each gate i in this branch, we assign outgoing label $i+\ell$ to the wire coming out of this gate and incoming labels $\ell+2i-1$ and $\ell+2i$ respectively to its two incoming wires. Therefore, we assume that the number of unique outgoing labels assigned in a branch are $G+\ell$, while the total number of unique incoming labels assigned in a branch are $W = 2G+\ell$. We present a slightly optimized version of the protocol described in the introduction, namely that only requires parties to sample 1 mask per wire, instead of 2 masks.

Let π be the mapping corresponding to a circuit C that maps incoming labels to the outgoing labels of each wire in C. For instance, $\pi(i)$ corresponds to the outgoing label of the wire with incoming label i. Let $\mathsf{C}_1,\ldots,\mathsf{C}_k$ be the circuit representations of the k branches and let $\{\pi_1,\ldots,\pi_k\}$ be the corresponding mappings associated with these branches. Finally, we assume that the circuits and inputs are defined over some field $\mathbb{F}$.

Protocol. The parties start by invoking $(func, f_1, x_1, \ldots, x_n, x_1, \ldots, x_\ell, b_1, \ldots, b_k)$ in $\mathscr{F}_{\mathsf{mpc}}$ on their original inputs $x_1,\ldots,x_n$, to obtain shares of inputs to the branching part $[x_1],\ldots,[x_\ell]$, where $|\ell|$ is the total input length and shares $[b_1],\ldots,[b_k]$, where $b_1 \ldots b_k$ is the unary representation of the index associated with the active branch. Given these shares, parties run the protocol presented in Figure 6.2. The output of this protocol is a secret sharing of the inputs to f_2 (i.e., the last part of the circuit). Let m be the length of these inputs. The parties finally invoke $(func, f_2, y_1, \ldots, y_m, \mathsf{out})$ and (out, out) in $\mathscr{F}_{\mathsf{mpc}}$ to learn the final output out.

Optimization. A naive implementation of the online phase in the above protocol will result in a round complexity that depends on the maximum number of gates in any particular branch. This can be improved to be proportional to the maximum multiplicative depth of any branch by using a simple optimization. For simplicity, lets assume that all branches have the same depth and each layer of each branch contains the same number of gates. We know that the gates on level ℓ only depend on the

[2] Our compiler can work with circuits that have gates with arbitrary fan-out. In our construction, it suffices to view such gates as having a single outgoing wire that acts as the incoming wire for multiple gates. Hence, we only assign a single label to the outgoing wire of each gate.

Semi-Honest Protocol

The protocol is described in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model. Parties have shares of inputs to the branches, i.e., $[x_1],\ldots,[x_\ell]$ and shares of a unary representation of the active branch, i.e., $[b_1],\ldots,[b_k]$.

- **Pre-processing Phase:**
  1. **Sample masks:** For each input and gate $g \in [\ell+G]$, parties invoke $(rand, \mathsf{mask}_g)$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain shares $[\mathsf{mask}_g]$. For each branch $m \in [k]$, let $[\overrightarrow{\mathsf{mask}_{\pi_m}}] = [\mathsf{mask}_{\pi_m(1)}] \| \ldots \| [\mathsf{mask}_{\pi_m(W)}]$.
  2. **Shares of zeros:** For each $w \in [W]$ and $i \in [n]$, parties invoke $(sharezero, X_{w,i})$ in $\mathscr{F}_{\mathsf{mpc}}$ to get shares $[X_{w,i}]$, where $X_{w,i} = 0$. For each $i \in [n]$, let $[\overrightarrow{X_i}] = [X_{1,i}] \| \ldots \| [X_{W,i}]$.
  3. **Pairwise OIP:** Each pair of parties P_R and P_S ($\forall \mathsf{R},\mathsf{S} \in [n]$) engage in a two-message semi-honest OIP as follows, where P_R acts as the receiver and P_S acts as the sender:
     - **Receiver:** P_R computes $(\rho, \mathsf{msg}_\mathsf{R}) \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, [b_1]_\mathsf{R}, \ldots, [b_k]_\mathsf{R})$ and sends msg_R to P_S.
     - **Sender:** P_S computes $\mathsf{msg}_\mathsf{S} \leftarrow \mathsf{OIP}_\mathsf{S}(1^\lambda, \mathsf{msg}_\mathsf{R}, [\overrightarrow{X_\mathsf{R}}]_\mathsf{S}, [\overrightarrow{\mathsf{mask}_{\pi_1}}]_\mathsf{S}, \ldots, [\overrightarrow{\mathsf{mask}_{\pi_k}}]_\mathsf{S})$ and sends msg_S to P_R.
     - **Output:** P_R computes $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}} \leftarrow \mathsf{OIP}_{\mathsf{out}}(\rho, \mathsf{msg}_\mathsf{R}, \mathsf{msg}_\mathsf{S})$.
  4. **Δ values:** Each party P_i (for $i \in [n]$) computes $[\overrightarrow{\Delta}]_i = \sum_{j\in[n]} \overrightarrow{\mathsf{share}_{j,i}}$, where $[\overrightarrow{\Delta}] = [\Delta_1] \| \ldots \| [\Delta_W]$.
- **Online Phase :**
  1. **Inputs:** For each input wire $i \in [\ell]$, parties compute $[u_i] = [x_i] + [\mathsf{mask}_i]$. and invoke (out, u_i) in $\mathscr{F}_{\mathsf{mpc}}$ to obtain u_i in the clear.
  2. **Circuit Evaluation:** For each gate $g \in [G]$, let $\mathsf{left} = \ell + 2g - 1$ and $\mathsf{right} = \ell + 2g$ be the incoming wire labels of its input wires. Let $\mathsf{type}_{m,g}$ be the gate type for gate g in C_m ($\forall m \in [k]$), where $\mathsf{type}_{m,g} = 0$ denotes an addition gate and $\mathsf{type}_{m,g} = 1$ denotes a multiplication gate. Parties compute the following using $\mathscr{F}_{\mathsf{mpc}}$:
     a) For $w \in \{\mathsf{left},\mathsf{right}\}$, compute $[y_w] = \sum_{m=1}^{k} \left(u_{\pi_m(w)} \cdot [b_m]\right) - [\Delta_w]$.
     b) Compute $[\mathsf{type}_g] = \sum_{m=1}^{k} \left(\mathsf{type}_{m,g} \cdot [b_m]\right)$
     c) Compute $[z_g] = [y_{\mathsf{left}}] + [y_{\mathsf{right}}] + [\mathsf{type}_g] \cdot ([y_{\mathsf{left}}] \cdot [y_{\mathsf{right}}] - [y_{\mathsf{left}}] - [y_{\mathsf{right}}])$.
     d) Compute $[u_{\ell+g}] = [z_g] + [\mathsf{mask}_{\ell+g}]$ and invoke (out, u_s) in $\mathscr{F}_{\mathsf{mpc}}$ to obtain u_s in the clear.
  3. **Output:** For each output gate g, compute $[z_g] = \sum_{m=1}^{k} \left(u_{\pi_m(w)} \cdot [b_m]\right) - [\Delta_w]$.

Figure 6.2: Semi-Honest Compiler

outgoing wires of gates on layers $< \ell$. We can therefore evaluate all the gates in a particular level in parallel. This simple idea can also be extended to the case where the branches have different depths and widths. In that case, let x_ℓ (and y_ℓ resp.) be the minimum (and maximum resp.) number of gates on level ℓ in any branch. We can evaluate the first x_ℓ gates in parallel. Then in the next round we can evaluate the $y_\ell - x_\ell + x_{\ell+1}$ gates in parallel. This ensures that the overall round complexity of the online phase will only depend on the depth of the branches.

Complexity Analysis. We now analyze the communication complexity of the above semi-honest protocol. If we use a rate-1 OIP, the communication complexity in the pre-processing phase is $O(n^2|C_{\max}| + n^2 k\lambda)$, where $|C_{\max}|$ is the size of the largest branch. In the online phase for each gate we perform both addition and multiplication and then choose between the two. As a result we perform 2 multiplications per gate. The communication complexity of the online phase is $O(2 \times \mathtt{CC}(|C_{\max}|))$, where $\mathtt{CC}(|C_{\max}|)$ is the communication complexity incurred upon evaluating $C_{\max}$ using the underlying MPC.

Overall, given the above protocol and optimizations, we obtain the following result.

Theorem 9 *Let λ be the security parameter and $\mathscr{F}$ be a function class consisting of functions of the form $f(\overrightarrow{x}) = f_2(f_{\mathsf{br}}(f_1(\overrightarrow{x})))$, where $f_{\mathsf{br}} := \{g_1, \ldots, g_k\}$ is a function consisting of k conditional branches, defined as $f_{\mathsf{br}}(i, \overrightarrow{x}) = g_i(\overrightarrow{x})$. Assuming the existence of a rate-1 two-message semi-honest secure OIP (see Definition 25), there exists an MPC protocol in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model (see Section 6.5) for computing any $f \in \mathscr{F}$ that achieves semi-honest security against an arbitrary number of corruptions and incurs a communication overhead of $O(n^2(k\lambda + |C_{\mathsf{max}}|))$.*

In Section 6.9, we show that a rate-1 two-message semi-honest secure OIP can be constructed from rate-1 linearly homomorphic encryption. Such encryptions are known [FV12, CL15, DJ01, PVW08] from a variety of assumptions including LWE, Ring LWE and DDH assumption.

Security

We now prove security of our semi-honest protocol. We start by describing the simulator and then proceed to argue indistinguishability between the real and ideal world executions.

Simulator. Let $\mathscr{A}$ be the adversary who corrupts a subset $\mathscr{I} \subset [n]$ of the parties and $\mathscr{H} = [n] \setminus \mathscr{I}$ be the set of honest parties. Let $\mathscr{S}_{\mathsf{R}}$ be the simulator associated with security of OIP against semi-honest receiver (see Definition 25). Given the output z and inputs $\{x_i\}_{i \in \mathscr{C}}$ of the corrupt parties the simulator proceeds as follows:

- **Computing f_1.** For each $i \in \mathscr{I}$, $j \in [\ell]$, sample random shares $[x_j]_i$ and for each $m \in [k]$, sample random shares $[b_m]_i$ and send all of these shares to the adversary.

- **Pre-processing Phase.**
  - For each $w \in [W]$ and $i \in \mathscr{I}$, sample random $[\mathsf{mask}_w]_i$ and send these values to the adversary.
  - For each $w \in [W]$ and $i \in [n]$ and $j \in \mathscr{I}$, sample random shares $[X_{w,i}]_j$ and sends to the adversary.
  - For each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), upon receiving a message msg_R from the adversary, sample a random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$, compute $\mathsf{msg}_\mathsf{S} \leftarrow \mathscr{S}_\mathsf{R}(1^\lambda, \mathsf{msg}_\mathsf{R}, \overrightarrow{V}_{\mathsf{R},\mathsf{S}})$ and send msg_S to the adversary on behalf of honest P_S.
  - For each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), set $[b_1]_\mathsf{R} = \ldots = [b_k]_\mathsf{R} = 0$, compute $\rho, \mathsf{msg}_\mathsf{R} \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, [b_1]_\mathsf{R}, \ldots, [b_k]_\mathsf{R})$ and send msg_R to the adversary on behalf of honest P_R. Also, set $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}} = [\overrightarrow{X_\mathsf{R}}]_\mathsf{S} + \sum_{i\in[k]} [b_i]_\mathsf{R} [\overrightarrow{\mathsf{mask}_{\pi_i}}]_\mathsf{S}$.
- **Online Phase.** In the online phase, the simulator mimcs the computation done by $\mathscr{F}_\mathsf{mpc}$. Recall from the description of $\mathscr{F}_\mathsf{mpc}$ (in Figure 6.1) that this only requires sending messages to the adversary whenever $(out, \cdot)$ is invoked. Since in the online phase, this is invoked on random values, the simulator can easily emulate this by sending a random value to the adversary for each such call.
- **Computing f_2.** Sends output z to the adversary.

Indistinguishability Argument. We argue indistinguishability via the following sequence of hybrids:

H_0 : This hybrid is identical to the real world execution.

H_1 : This hybrid is very similar to the previous hybrid except that in the preprocessing phase for each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), we change the way msg_R and $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}}$ are computed:

- $\rho, \mathsf{msg}_\mathsf{R} \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, 0 \ldots, 0)$.
- $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}} = [\overrightarrow{X_\mathsf{R}}]_\mathsf{S} + \sum_{i\in[k]} [b_i]_\mathsf{R} [\overrightarrow{\mathsf{mask}_{\pi_i}}]_\mathsf{S}$.

For indistinguishability between hybrids H_0 and H_1, we consider a sequence of sub-hybrids, where we change the way msg_R and $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}}$ are computed for each pair P_R and P_S (where $\mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), one hybrid at a time. In terms of the view of the adversary, the only change in each of these sub-hybrids is in the way msg_R is computed for one pair $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$. As a result, indistinguishability between each consecutive pair of sub-hybrids follows the security of OIP against a semi-honest sender and by transitivity, it holds that H_0 and H_1 are indistinguishable.

H_2 : This hybrid is very similar to the previous hybrid except that in the preprocessing phase, for each pair of parties P_{R} and P_{S} ($\forall \mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), we compute $\mathsf{msg}_{\mathsf{S}} \leftarrow \mathscr{S}_{\mathsf{R}}(1^{\lambda}, \mathsf{msg}_{\mathsf{R}}, \overrightarrow{V}_{\mathsf{R},\mathsf{S}})$, using some random share $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$.

For indistinguishability between hybrids H_1 and H_2, we consider a sequence of sub-hybirds, where we change the way $\mathsf{msg}_{\mathsf{S}}$ is computed for each pair $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$, one hybrid at a time.

The only difference between any two consecutive pairs of these sub-hybrids is that in one we compute $\mathsf{msg}_{\mathsf{S}}$ using a random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$ and the simulator for some receiver R and sender S, while in the other $\mathsf{msg}_{\mathsf{S}}$ is computed honestly and the output that the receiver gets is $[\overrightarrow{X_{\mathsf{R}}}]_{\mathsf{S}} + \sum_{m \in [k]} [b_m]_{\mathsf{R}} [\overrightarrow{\mathsf{mask}_{\pi_m}}]_{\mathsf{S}}$. Since $[\overrightarrow{X_{\mathsf{R}}}]_{\mathsf{S}}$ is a random vector of shares, this output is identically distributed to $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$. Given this output indistinguishability between a simulated message $\mathsf{msg}_{\mathsf{S}}$, and an honestly computed message $\mathsf{msg}_{\mathsf{S}}$ follows from security of OIP against a semi-honest receiver. As a result, this sub-hybrid is indistinguishable from its previous hybrid and by transitivity, it holds that H_1 and H_2 are indistinguishable.

H_3 : This hybrid is identical to the simulator description.

Indistinguishability between hybrids H_2 and H_3 follows from semi-honest security of the underlying MPC protocol.

6.7 Non-Constant Round Maliciously Secure Branching MPC

In this section, we present our maliciously secure compiler for distributed computation of a branching circuit. We borrow notations from the previous section. As discussed in the introduction, the basic outline of our maliciously secure protocol remains the same except that we now use a two-message OIP that is secure against malicious receivers. Also, in order to ensure that the sender behaves honestly, we make use of non-interactive commitments.

Protocol. Similar to the semi-honest protocol, the parties start by invoking $(func, f_1, , x_1, \ldots, x_n, , x_1, \ldots, x_\ell, b_1, \ldots, b_k)$ in $\mathscr{F}_{\mathsf{mpc}}$ on their original inputs $x_1, \ldots, x_n$, to obtain shares of inputs to the branching part $[x_1], \ldots, [x_\ell]$, where $|\ell|$ is the total input length and shares $[b_1], \ldots, [b_k]$, where $b_1 \ldots b_k$ is the unary representation of the index associated with the active branch. Given these shares, the parties run the protocol presented in Figures 6.3 and 6.4. The output of this protocol is a secret sharing of the inputs to f_2 (i.e., the last part of the circuit). Let m be the length of these inputs. The parties finally invoke $(func, f_2, y_1, \ldots, y_m, \mathsf{out})$ and (out, out) in $\mathscr{F}_{\mathsf{mpc}}$ to learn the final output out.

Complexity Analysis. If we use a rate-1 OIP, the communication complexity in the preprocessing phase is $O(\delta \times n^2 |C_{\max}| + n^2 k \lambda)$, where $|C_{\max}|$ is the size of the largest branch and $\delta = \kappa / 0.311$. The online phase is repeated for each $q \in [\delta] \setminus Z$,

as a result, the communication complexity of the online phase is $O(\delta \times \mathtt{CC}(|C_{\max}|))$, where $\mathtt{CC}(|C_{\max}|)$ is the communication complexity incured upon evalutaing $C_{\max}$ using the underlying MPC.

Overall, the above protocol gives us the following result.

Theorem 10 *Let λ be the computational security parameter and κ be the statistical security parameter. Let $\mathscr{F}$ be a function class consisting of functions of the form $f(\overrightarrow{x}) = f_2(f_{\mathsf{br}}(f_1(\overrightarrow{x})))$, where $f_{\mathsf{br}} := \{g_1, \ldots, g_k\}$ is a function consisting of k conditional branches, defined as $f_{\mathsf{br}}(i, \overrightarrow{x}) = g_i(\overrightarrow{x})$. Assuming the existence of a rate-1 two-message OIP secure against a malicious receiver (see Definition 25), there exists an MPC protocol in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model (see Section 6.5) for computing any $f \in \mathscr{F}$ that achieves security with abort against an arbitrary number of malicious corruptions and incurs a communication overhead of $O(n^2(k\lambda + \kappa|C_{\mathsf{max}}|))$.*

Rate-1 two-message OIP secure against a malicious receiver can be built from rate-1 linearly homomorphic encryption and non-interactive zero-knowledge.

Security

We now prove security of our malicious protocol. We start by describing the simulator and then proceed to argue indistinguishability between the view of the adversary in the real protocol execution and the view generated by the simulator.

Simulator. Let $\mathscr{I} \subset [n]$ be the set of corrupt parties and $\mathscr{H} = [n] \setminus \mathscr{I}$ be the set of honest parties. Let $\mathscr{S}_{\mathsf{R}} = (\mathscr{S}^1_{\mathsf{R}}, \mathscr{S}^2_{\mathsf{R}})$ be the simulators corresponding to the malcious receiver security of OIP. Given the output z and inputs $\{x_i\}_{i \in \mathscr{C}}$ of the corrupt parties the simulator proceeds as follows:

- **Computing f_1.** When the adversary sends its inputs to $\mathscr{F}_{\mathsf{mpc}}$ while invoking $func$, the simulator receives them and queries the ideal functionality on these inputs to get the final output. For each $i \in \mathscr{I}$, $j \in [\ell]$, the simulator samples random share $[x_j]_i$ and for each $m \in [k]$, it samples random share $[b_m]_i$ and sends these shares to the adversary.

- **Pre-processing Phase.**

  - For each $q \in [\delta]$, $w \in [W]$ and $i \in \mathscr{I}$, the simulator samples random shares $[\mathsf{mask}^q_w]_i$ and sends these values to the adversary.
  - For each $q \in [\delta]$, $w \in [W]$ and $i \in [n]$ and $j \in \mathscr{I}$, the simulator samples random shares $[X^q_{w,i}]_j$ and sends to the adversary.
  - It samples a random subset $Z \subset [\delta]$ of size $\delta/2$.
  - For each pair of parties P_{R} and P_{S} ($\forall \mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), the simulator sets $[b_1]_{\mathsf{R}} = \ldots = [b_k]_{\mathsf{R}} = 0$, computes $\rho, \mathsf{msg}_{\mathsf{R} \to \mathsf{S}} \leftarrow \mathsf{OIP}_{\mathsf{R}}(1^\lambda, [b_1]_{\mathsf{R}}, \ldots, [b_k]_{\mathsf{R}})$ and sends $\mathsf{msg}_{\mathsf{R}}$ to the adversary on behalf of honest P_{R}.

– For each pair of parties P_{R} and P_{S} ($\forall \mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), the simulator proceeds as follows:

∗ For each $q \in [\delta]$, it samples random values $r_0^q, \ldots, r_k^q \in \mathbb{F}^{k+1}$.

∗ For each $q \in Z$, sample random shares $[\overrightarrow{X_{\mathsf{R}}^q}]_{\mathsf{S}}, [\overrightarrow{\mathsf{mask}_{\pi_1}^q}], \ldots, [\overrightarrow{\mathsf{mask}_{\pi_k}^q}]$ and compute

$$\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}^q \leftarrow \mathsf{OIP}_{\mathsf{S}}(1^\lambda, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}}, [\overrightarrow{X_{\mathsf{R}}^q}]_{\mathsf{S}} \| r_0^q, [\overrightarrow{\mathsf{mask}_{\pi_1}^q}]_{\mathsf{S}} \| r_1^q, \ldots, [\overrightarrow{\mathsf{mask}_{\pi_k}^q}]_{\mathsf{S}} \| r_k^q; \rho_{\mathsf{R},\mathsf{S}}^{\mathsf{S},q})$$

$$c_{\mathsf{R},\mathsf{S}}^q \leftarrow \mathsf{Commit}([\overrightarrow{X_{\mathsf{R}}^q}]_{\mathsf{S}} \| r_0^q, [\overrightarrow{\mathsf{mask}_{\pi_1}^q}]_{\mathsf{S}} \| r_1^q, \ldots, [\overrightarrow{\mathsf{mask}_{\pi_k}^q}]_{\mathsf{S}} \| r_k^q, \rho_{\mathsf{R},\mathsf{S}}^{\mathsf{S},q}; \rho_{\mathsf{R},\mathsf{S}}^{c,q})$$

.

∗ For each $q \in [\delta] \setminus Z$, the simulator samples a random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}^q$ and computes $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}^q \leftarrow \mathscr{S}_{\mathsf{R}}^2(1^\lambda, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}}, \overrightarrow{V}_{\mathsf{R},\mathsf{S}}^q)$ and $c_{\mathsf{R},\mathsf{S}}^q \leftarrow \mathsf{Commit}(0; \rho_{\mathsf{R},\mathsf{S}}^{c,q})$.

∗ For each $q \in [\delta]$, it sends $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}^q$ and $c_{\mathsf{R},\mathsf{S}}^q$ to the adversary on behalf of an honest P_{S}.

∗ **Receiver Consistency Check:** For each $q \in Z$, upon receiving $\mathsf{mac}_{\mathsf{R},\mathsf{S}}^\delta$, the simulator runs $\mathscr{S}_{\mathsf{R}}^1$ on $\mathsf{msg}_{\mathsf{R}\to\mathsf{S}}$ to extract $[b_1]'_{\mathsf{R}}, \ldots, [b_k]'_{\mathsf{R}}$. Check if for each $m \in [k]$, $[b_m]'_{\mathsf{R}} = [b_m]_{\mathsf{R}}$ and if $\mathsf{mac}_{\mathsf{R},\mathsf{S}}^\delta = r_0^q + \sum_{m\in[k]} [b_m]_{\mathsf{R}} r_m^q$. If both these checks succeed, output 1 when the parties invoke the *checkzero* function, else output 0 and the simulator signals the ideal functionality to send abort to the honest parties and aborts the protocol.

– **Sender Consistency Check:** The simulator sends Z to the adversary. For each $q \in Z$ send all shares of $\mathsf{mask}_{\pi_1}^q \ldots, \mathsf{mask}_{\pi_k}^q$ and $X_{w,i}^q$ used in the OIP instances to the adversary.

∗ Send all shares of $\mathsf{mask}_{\pi_1}^q \ldots, \mathsf{mask}_{\pi_k}^q$ and $X_{w,i}^q$ used in the OIP instances to the adversary.

∗ For each pair of parties P_{R} and P_{S} (where $\mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), the simulator checks if all the messages and commitments were honestly computed. If not, it signals the ideal functionality to send abort to the honest parties and aborts the protocol.

∗ For each pair of parties P_{R} and P_{S} (where $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), the simulator proceeds exactly as in the real protocol.

• **Online Phase.** In the online phase, the simulator mimics the computation done by $\mathscr{F}_{\mathsf{mpc}}$, except that it does not compute the majority function. Instead it sends random shares for each output gate to the adversary.

For all other steps in the online phase, based on the description of $\mathscr{F}_{\mathsf{mpc}}$ (in Figure 6.1), the simulator only needs to send messages to the adversary whenever $(out, \cdot)$ is invoked. Since in the online phase, this is only invoked on random values, the simulator can easily emulate this by sending a random value to the adversary for each such call.

- **Computing f_2.** Outputs the output received from the ideal functionality in the first step to the adversary and send continue to the ideal functionality to signal that the honest parties can learn the output.

Indistinguishability Argument. We argue indistinguishability between the real and ideal executions, via the following sequence of hybrids:

H_0 : This hybrid is identical to the real world execution.

H_1 : This hybrid is very similar to the previous one, except that the subset Z is sampled before the parties engage in the pairwise OIP protocols, but is revealed to the parties only during the sender consistency checks.

Hybrids H_0 and H_1 are trivially indistinguishable.

H_2 : This hybrid is very similar to the previous one, except that for each $q \in [\delta] \setminus Z$ and each pair of parties P_R and P_S (where $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), we compute $c^q_{\mathsf{R},\mathsf{S}} \leftarrow \mathsf{Commit}(0; \rho^{c,q}_{\mathsf{R},\mathsf{S}})$.

Indistinguishability between hybrids H_1 and H_2 follows from a sequence of sub-hybrids, where we change the way $c^q_{\mathsf{R},\mathsf{S}}$ is computed for $q \in [\delta] \setminus Z$ and each pair P_R and P_S (where $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), one hybrid at a time. Indistinguishability between each consecutive pair of sub-hybrids follows from the hiding property of the commitment scheme and by transitivity it follows that H_1 and H_2 are indistinguishable.

H_3 : This hybrid is very similar to the previous hybrid except that in the preprocessing phase, for each $q \in [\delta] \setminus Z$ and for each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), we compute $\mathsf{msg}^q_{\mathsf{S}\to\mathsf{R}} \leftarrow \mathscr{S}^2_\mathsf{R}(1^\lambda, \mathsf{msg}_\mathsf{R}, \overrightarrow{V}^q_{\mathsf{R},\mathsf{S}})$, for some random vector of shares $\overrightarrow{V}^q_{\mathsf{R},\mathsf{S}}$.

For indistinguishability between hybrids H_2 and H_3, we consider a sequence of sub-hybirds, where we change the way $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}$ is computed for each $q \in [\delta] \setminus Z$ and each pair $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$, one hybrid at a time.

The only difference between any two consecutive pairs of these sub-hybrids is that in one we compute $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}$ using a random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$ and the simulator for some $q \in [\delta] \setminus Z$ and some receiver R and sender S, while in the other, $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}$ is computed honestly and the output that the receiver gets is $[\overrightarrow{X_\mathsf{R}}]_\mathsf{S} + \sum_{m\in[k]} [b_m]_\mathsf{R} [\overrightarrow{\mathsf{mask}_{\pi_m}}]_\mathsf{S}$. Since $[\overrightarrow{X_\mathsf{R}}]_\mathsf{S}$ is a random vector of shares, this output is identically distributed to $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$. Given this output, indistinguishability between a simulated message $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}$, and an honestly computed message $\mathsf{msg}_{\mathsf{S}\to\mathsf{R}}$ follows from security of OIP against a malicious receivers. As a result, this sub-hybrid is indistinguishable from its previous hybrid and by transitivity, it holds that H_2 and H_3

H_4 : This hybrid is very similar to the previous hybrid except that in the preprocessing phase for each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), we change the way msg_R and $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}}$:

- $\rho, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}} \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, 0\ldots,0)$.
- For each $q \in [\delta] \setminus Z$, $\overrightarrow{\mathsf{share}^q_{\mathsf{R},\mathsf{S}}} = [\overrightarrow{X^q_\mathsf{R}}]_\mathsf{S} + \sum_{i\in[k]} [b_i]_\mathsf{R} [\overrightarrow{\mathsf{mask}^q_{\pi_i}}]_\mathsf{S}$.

For indistinguishability between hybrids H_3 and H_4, we consider a sequence of sub-hybrids, where we change the way $\mathsf{msg}_{\mathsf{R}\to\mathsf{S}}$ and $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}}$ are computed for each $q \in [\delta] \setminus Z$ and each pair P_R and P_S (where $\mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), one hybrid at a time. In terms of the view of the adversary, the only change in each of these sub-hybrids is in the way $\mathsf{msg}_{\mathsf{R}\to\mathsf{S}}$ is computed for one pair $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$ and some $q \in [\delta] \setminus Z$. As a result, indistinguishability between each consecutive pair of sub-hybrids follows the security of OIP against a semi-honest sender and by transitivity, it holds that H_3 and H_4 are indistinguishable.

H_5 : This hybrid is identical to the simulator description.

The main difference between H_4 and H_5 is that in H_5, simulator emulates the $\mathscr{F}_\mathsf{mpc}$ functionality and hence, the way the output of the honest parties is computed differs in the two protocols. In particular, in H_4, we take a majority of all the runs of the online phase and only if there exists an output that appears $> \delta/4$ times, do we open the ouput. In H_5, the simulator simply checks if all the opened commitments are consistent to consider the output. Let $\mathsf{noAbort}$ denote an event where in our cut-and-choose step, all instances in Z are valid. Also let badMaj denote the event where $> \delta/4$ instances in the remaining set $[\delta] \setminus Z$ are invalid. To argue indistinguishability between the output of honest parties in the hybrids H_4 and H_5, we start by recalling Claim 4.3 from [LP11]. At a high level, this claim essentially states that the probability that $> \delta/4$ instances are invalid and neither gets caught in the opening phase of the cut-and-choose protocol, is $1/2^{0.311\delta}$. More formally

Claim 1 *For every $\delta \in \mathbb{N}$, it holds that*

$$\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] = \frac{\binom{\frac{3\delta}{4}+1}{\frac{\delta}{2}+1}}{\binom{\delta}{\delta/2}} < \frac{1}{2^{\frac{\delta}{4}-1}}$$

and for large enough s (depending on Stirling's approximation), it holds that $\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] = \frac{1}{2^{0.311\delta}}$

From the binding property of the commitment scheme, it follows that if any instance in Z is invalid, it will get caught in both H_4 and H_5. And if all the checks performed on these instances in Z succeed, then indeed, they are all valid/consistent and it is a $\mathsf{noAbort}$ event. In this case, in hybrid H_5, the

simulator signals the ideal functionality to send the correct output to the honest parties. Therefore, the output of the honest parties in this case will differ from their output in hybrid H_4 only when $\mathsf{noAbort} \wedge \mathsf{badMaj}$ happens. For $\delta = \kappa/0.311$, this only happens with exponentially small probability and hence the output of the honest parties is indistinguishable in the two hybrids H_4 and H_5. Indistinguishability between the view of the adversary in the two hybrids follows from malicious security of the underlying MPC protcol.

6.8 Constant Round Semi-Honest Branching MPC

In this section we present our constant round semi-honest protocol for distributed computation of a branching circuit.

As discussed in the technical overview, we encrypt keys for the output wires of each gate during garbling using the help of a random function instantiated using the decisional Ring LWE (RLWE) assumption. Let $p = 2N+1$ be a prime, where N, called the dimension or security parameter, is a power of 2. Let $\mathscr{R}_p = \mathbb{Z}_p[X]/(X^N+1)$ be the polynomial ring over $\mathbb{Z}_p$ modulo X^N+1. We start by recalling the decisional RLWE assumption stated by Ben-Efraim et al. [BLO17].

Definition 26 (Decisional Ring LWE Problem) *Any non-uniform* PPT *adversary cannot distinguish between* $\{(a_i,b_i)\}_{i\in I}$ *and* $\{(a_i,a_i\cdot k+\delta_i)\}_{i\in I}$ *with non-negligible probability where* $\{a_i\}_{i\in I}$, $\{b_i\}_{i\in I}$ *and* k *are chosen uniformly at random from* $\mathscr{R}_p$ *and the coeffecients of* $\{e_i\}_{i\in I}$ *are sampled from* χ*, a spherical Gaussian distribution.*

By transforming to the Hermite normal form, the decisional RLWE assumption also holds if the key k is chosen from a spherical Gaussian distribution. Similar to the construction of Ben-Efraim et al. [BLO17], sampling both the key and error from χ is key to eliminating the error during decryption in the evaluation phase of our protocol. Specifically, if the mean of the Gaussian distribution χ is $\frac{\sqrt{p}}{2}$ and the standard deviation is sufficiently small, a sample is *not* in the range $[0,\sqrt{p}]$ with negligible probability. Thus, dividing by $\sqrt{p}$ during decryption removes the error and recovers the message if the message was multiplied by $\sqrt{p}$ during encryption.

While we use new public random elements $A_g^{u,v}$ from the ring for every RLWE expansion in our protocol, [BLO17] shows that $8\cdot f_{\mathsf{out}}$ uniformly random and public elements from the ring suffice, where f_{out} is the maximal fan-out of the circuit, as long as ciphertexts for gates that share inputs wires are computed using distinct sets of elements. Similar to [BLO17], $8\cdot f_{\mathsf{out}}$ must be less than the bound on the number of RLWE samples $|I|$, for security to hold. We refer the reader to [LPR10, LPR13] for more details about the decisional RLWE assumption.

Our protocol follows the BMR approach which involves sampling a pair of keys k_w^0, k_w^1 for each wire w in the circuit. A garbled table is then constructed for each gate such that the key corresponding to the value on the output wire is encrypted using the keys corresponding to the input values. Since the position of each ciphertext in the garbled table leaks information about its plaintext, a private random mask bit

$\gamma_w \in \{0,1\}$ is sampled for each wire w and the masks for the input wires are used to permute the rows of the garbled table for each gate. Let the external value β_w on a wire be the plaintext value ρ_w on the wire masked with the mask γ_w i.e., $\beta_w = \rho_w \oplus \gamma_w$. Then, the masks on the input wires are used to permute the rows of the garbled table such that the external values on the input wires can be used to index into the required row of the garbled table. Thus, to ensure that parties decrypt the correct row when evaluating the circuit, the mask for the output wire has to also be included in the ciphertext for each row. We use the approach of Ben-Efraim et al. [BLO17], where the last coordinate of the keys k_w^0, k_w^1 for each wire are set to 0, which slightly reduces security, and the external value is embedded into this coordinate during encryption. We use $k\|e$ to denote that the bit e was embedded in the last coordinate of the key k.

The garbling phase is presented in Figure 6.5 and the evaluation phase is presented in Figure 6.6. We adopt the same notation as the semi-honest protocol presented in Figure 6.2. If ℓ be the number of input wires to the branching part of the function, we set the incoming and outgoing labels for these wires to be $1,\ldots,\ell$. For each gate g we set the outgoing wire label to be $\ell+g$, the left incoming wire label to be $\ell+2g-1$ and the right incoming wire label to be $\ell+2g$. We also let π_m for each $m \in [k]$ to be the mapping that maps incoming labels to the outgoing labels of each wire for the m-th branch.

Finally, we remark that we require $\mathscr{F}_{\mathsf{mpc}}$ to run in constant number of rounds for constant depth circuits to ensure that our protocol has constant number of rounds. This is true for most secret sharing based protocols that evaluate the circuit in a gate-by-gate manner.

Complexity Analysis. We now analyze the communication complexity of the above constant round semi-honest protocol. We assume that the size of the ring $\mathscr{R}_p$ is in $O(\lambda)$. If we use a constant rate semi-honest secure OIP, the communication complexity in the garbling phase is $O(n^2|C_{\max}| + n^2 k\lambda\,\mathtt{CC}(\lambda|C_{\max}|))$, where $|C_{\max}|$ is the size of the largest branch and $\mathtt{CC}(\lambda|C_{\max}|)$ is the communication complexity incurred upon evaluating $C_{\max}$ using the underlying MPC. In the evaluation phase, the communication cost incurred is for reconstructing $O(\lambda|C_{\max}|)$ shares corresponding to the garbling material.

Overall, given the above protocol and optimizations, we obtain the following result.

Theorem 11 *Let λ be the security parameter and $\mathscr{F}$ be a function class consisting of functions of the form $f(\overrightarrow{x}) = f_2(f_{\mathsf{br}}(f_1(\overrightarrow{x})))$, where $f_{\mathsf{br}} := \{g_1,\ldots,g_k\}$ is a function consisting of k conditional branches, defined as $f_{\mathsf{br}}(i,\overrightarrow{x}) = g_i(\overrightarrow{x})$. Assuming that a rate-1 two-message semi-honest secure OIP exists (see Definition 25) and that the decisional RLWE problem holds (see Definition 26), there exists a constant-round MPC protocol in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model (see Section 6.5) for computing any $f \in \mathscr{F}$ that achieves semi-honest security against an arbitrary number of corruptions and incurs a communication overhead of $O(n^2\lambda(k+|C_{\mathsf{max}}|))$.*

Note that if we instatiate the rate-1 two-message semi-honest secure OIP using RLWE-based linearly homomorphic encryption, then the above theorem yields a protocol that only relies on the hardness of the decisional RLWE.

Security

We now prove security of our constant round semi-honest protocol. We start by describing the simulator and then proceed to argue indistinguishability between the real and ideal world executions.

Simulator. Let $\mathscr{A}$ be the adversary who corrupts a subset $\mathscr{I} \subset [n]$ of the parties and $\mathscr{H} = [n] \setminus \mathscr{I}$ be the set of honest parties. Let $\mathscr{S}_{\mathsf{R}}$ be the simulator associated with the semi-honest security against receiver of the OIP. Given the output z and inputs $\{x_i\}_{i\in\mathscr{C}}$ of the corrupt parties the simulator proceeds as follows:

- **Computing f_1.** For each $i \in \mathscr{I}$, $j \in [\ell]$, the simulator samples random share $[x_j]_i$ and for each $m \in [k]$, it samples random share $[b_m]_i$ and sends all these shares to the adversary.
- **Garbling phase.**
  - For each input and gate $g \in [\ell + G]$, the simulator samples random $[\gamma_g]_i \leftarrow \mathscr{R}_p$ for each $i \in \mathscr{I}$ and sends it to the adversary.
  - For each $j \in [n]$, $w \in [W + 4G]$, and $i \in \mathscr{I}$, the simulator samples random $[X_{j,w}]_i \leftarrow \mathscr{R}_p$ and sends it to the adversary.
  - For each pair of parties P_{R} and P_{S} ($\forall \mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), the simulator sets $[b_1]_{\mathsf{R}} = \ldots = [b_k]_{\mathsf{R}} = 0$, computes $\rho, \mathsf{msg}_{\mathsf{R}} \leftarrow \mathsf{OIP}_{\mathsf{R}}(1^\lambda, [b_1]_{\mathsf{R}}, \ldots, [b_k]_{\mathsf{R}})$ and sends $\mathsf{msg}_{\mathsf{R}}$ to the adversary on behalf of honest P_{R}.
  - For each pair of parties P_{R} and P_{S} ($\forall \mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), upon receiving a message $\mathsf{msg}_{\mathsf{R}}$ from the adversary, the simulator samples a random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$ and computes $\mathsf{msg}_{\mathsf{S}} \leftarrow \mathscr{S}_{\mathsf{R}}(1^\lambda, \mathsf{msg}_{\mathsf{R}}, \overrightarrow{V}_{\mathsf{R},\mathsf{S}})$. It sends $\mathsf{msg}_{\mathsf{S}}$ to the adversary on behalf of honest P_{S}.
  - When garbling the active branch, the simulator mimics the computation done by $\mathscr{F}_{\mathsf{mpc}}$.
- **Evaluation phase.**
  - For each input wire $w \in [\ell]$, the simulator samples random $\beta_w \leftarrow \{0,1\}$, and sends it to the adversary.
  - For each input wire $w \in [\ell]$, the simulator samples random $k_w^{\beta_w} \leftarrow \mathscr{R}_p$, and also sends it to the adversary.
  - For each $g \in [G]$ and $u, v \in \{0,1\}$, the simulator samples random $C_g^{u,v} \leftarrow \mathscr{R}_p$, randomly sets the last coordinate to either 0 or 1 and sends it to the adversary.

Indistinguishability Argument. We argue indistinguishability via the following sequence of hybrids:

H_0 : This hybrid is identical to the real world execution.

H_1 : This hybrid is very similar to the previous hybrid except that in the garbling phase for each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), we change the way msg_R and $\overrightarrow{\mathsf{share}}_{\mathsf{R},\mathsf{share}}$ are computed:

- $\rho, \mathsf{msg}_\mathsf{R} \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, 0\ldots,0)$.
- $\overrightarrow{\mathsf{share}}_{\mathsf{R},\mathsf{S}} = [\overrightarrow{X_\mathsf{R}}]_\mathsf{S} + \sum_{i\in[k]} [b_i]_\mathsf{R} [\overrightarrow{x_i}]_\mathsf{S}$.

For indistinguishability between hybrids H_0 and H_1, we consider a sequence of sub-hybrids, where we change the way msg_R and $\overrightarrow{\mathsf{share}}_{\mathsf{R},\mathsf{S}}$ are computed for each pair P_R and P_S (where $\mathsf{R} \in \mathscr{H}, \mathsf{S} \in \mathscr{I}$), one hybrid at a time. In terms of the view of the adversary, the only change in each of these sub-hybrids is in the way msg_R is computed for one pair $\mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$. As a result, indistinguishability between each consecutive pair of sub-hybrids follows the security of OIP against a semi-honest sender and by transitivity, it holds that H_0 and H_1 are indistinguishable.

H_2 : This hybrid is very similar to the previous hybrid except that in the garbling phase, for each pair of parties P_R and P_S ($\forall \mathsf{R} \in \mathscr{I}, \mathsf{S} \in \mathscr{H}$), we compute $\mathsf{msg}_\mathsf{S} \leftarrow \mathscr{S}_\mathsf{R}(1^\lambda, \mathsf{msg}_\mathsf{R}, \overrightarrow{V}_{\mathsf{R},\mathsf{S}})$ for some random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$.

The only difference between any two consecutive pairs of these sub-hybrids is that in one we compute msg_S using a random vector of shares $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$ and the simulator for some receiver R and sender S, while in the other msg_S is computed honestly and the output that the receiver gets is $[\overrightarrow{X_\mathsf{R}}]_\mathsf{S} + \sum_{i\in[k]} [b_i]_\mathsf{R} [\overrightarrow{x_i}]_\mathsf{S}$. Since $[\overrightarrow{X_\mathsf{R}}]_\mathsf{S}$ is a random vector of shares, this output is identically distributed to $\overrightarrow{V}_{\mathsf{R},\mathsf{S}}$. Given this output indistinguishability between a simulated message msg_S, and an honestly computed message msg_S follows from security of OIP against a semi-honest receiver. As a result, this sub-hybrid is indistinguishable from its previous hybrid and by transitivity, it holds that H_1 and H_2 are indistinguishable.

H_3 : This hybrid is similar to the previous hybrid except that $C_g^{u,v}$ is sampled uniformly at random from $\mathscr{R}_p$ for all $g \in [G]$ and $u, v \in \{0,1\}$.

Indistinguishability between hybrids H_2 and H_3 follows from a sequence of sub-hybrids, each relying on the decisional RLWE hardness assumption, where we change $C_g^{u,v}$ one at a time.

H_4 : This hybrid is identical to the simulator description.

Indistinguishability between hybrids H_3 and H_4 follows from the semi-honest security of the underlying MPC protocol.

6.9 OIP from Linearly Homomorphic Encryption

In this section, we show how to construct OIPs from linearly homomorphic encryption.

Linearly Homomorphic Encryption

We start by recalling the definition of linearly homomorphic encryption.

Definition 27 (Linearly Homomorphic Encryption) *A linearly homomorphic encryption scheme over a message space $\mathscr{M}$ is defined by a tuple of 4 PPT algorithms* $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathit{Eval})$ *as follows:*

- $(pk, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$*: On input the security parameter λ, the key generation algorithm outputs a public key pk and a secret key* sk*.*
- $c \leftarrow \mathsf{Enc}_{(}pk, \mathsf{x})$*: On input the public key pk and a message $x \in \mathscr{M}$, the encryption algorithm outputs a ciphertext c.*
- $x \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$*: On input the secret key* sk *and the ciphertext c, the decryption algorithm outputs a message $m \in \mathscr{M}$.*
- $c_L \leftarrow \mathit{Eval}(pk, L, c_1, \ldots, c_k)$ *On input the public key pk, a set of ciphertexts $c_1 \ldots, c_k$ and a linear function $L : \mathscr{M}^k \to \mathscr{M}$, the evaluation algorithm outputs another ciphertext c_L.*

We proceed to define three main properties of a linearly homomorphic encryption scheme: correctness, homomorphism *and* privacy.

- ***Correctness:*** *Let $(pk, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$. Then for any $x \in \mathscr{M}$, it holds that:*

$$\Pr\left[\mathsf{Dec}\left(\mathsf{sk}, \mathsf{Enc}_{(}pk, \mathsf{x})\right) = x\right] = 1$$

- ***Homomorphism:*** *Let $(pk, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$. Then for any $x_1, \ldots, x_k \in \mathscr{M}^k$ and any linear function $L : \mathscr{M}^k \to \mathscr{M}$, it holds that:*

$$\Pr\left[\mathsf{Dec}\left(\mathsf{sk}, \mathit{Eval}\left(pk, L, \{\mathsf{Enc}_{(}pk, \mathsf{x_i})\}_{\mathsf{i}\in[\mathsf{k}]}\right)\right) = L(x_1, \ldots, x_k)\right] = 1$$

- ***Circuit Privacy:*** *There exists a PPT simulator S, such that for every PPT adversary $\mathscr{A}$ with inputs $x_1, \ldots, x_k \in \mathscr{M}^k$ and any linear function $L : \mathscr{M}^k \to \mathscr{M}$ the following distributions are computationally indistinguishable:*

$$\mathit{Eval}\left(pk, L, \{c_i\}_{i\in[k]}\right) \approx_c S(1^\lambda, pk, L(\{x_i\}_{i\in[k]})),$$

where $(pk, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$ and $c_i \leftarrow \mathsf{Enc}_{(}pk, \mathsf{x_i})$ for each $i \in [k]$ were generated honestly by $\mathscr{A}$.

- ***Privacy:*** *For all n.u. PPT adversaries $\mathscr{A}$, there exists a negligible function $\mu(\cdot)$ such that:*

$$\Pr\left[\begin{array}{ll}(pk,\mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda), & \\ (x_0,x_1) \leftarrow \mathscr{A}(1^\lambda,pk), & :\mathscr{A}(\mathsf{Enc}_(\mathsf{sk},\mathsf{x_b})) = \mathsf{b} \\ b \xleftarrow{\$} \{0,1\} & \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

where $|x_0| = |x_1|$.

Such linearly homomorphic encryption can be obtained from a variety of assumptions [FV12, CL15, DJ01, PVW08]. In our implementation we use a variant of the BFV scheme [FV12].

Constructing OIP

We now describe a simple construction of a semi-honest OIP using linearly homomorphic encryption.

Lemma 10 *Let* $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathit{Eval})$ *be a linearly homomorphic encryption, then the construction in Figure 6.7 is a two-message semi-honest oblivious inner product protocol.*

Proof 16 *Correctness of this construction follows trivially from the correctness and homomorphic property of the underlying linearly homomorphic encryption scheme. Security against semi-honest sender follows from privacy of the encryption scheme and security against semi-honest receiver follows from circuit privacy.*

Remark. Note that if the linearly homomorphic encryption scheme has a constant rate, then the length of the sender message in the above construction of OIP only depends on the length of the output and not on the inputs of the sender. Also, the above construction is only semi-honest secure. For our maliciously secure MPC, we also require an OIP that is secure against a malicious receiver. Such an OIP can be easily constructed by attaching non-interactive zero-knowledge proofs of knowledge to the receiver messages.

6.10 Implementation

We implement and benchmark our semi-honest non-constant round protocol from Section 6.6. The code is publicly available at `https://github.com/rot256/research-branching-mpc`. In addition to the code and instructions used for benchmarking, the repository also contains the raw data used in this paper and scripts used to create the plots.

How We Benchmark

Underlaying MPC. We implement our semi-honest compiler on top of two different multi-party computation protocols.

1. *Quadratic Dependence on the Number of Parties.* A semi-honest variant of MASCOT [KOS16] (MASCOT without sacrificing and message authentication codes) over the prime field $\mathbb{F}_{2^{16}+1} = \mathbb{Z}/(\texttt{0x10001}\ \mathbb{Z})$ provided by MP-SPDZ [Kel20] (called "`semi-party.x`"). We simply invoke the MP-SPDZ implementation as a black-box: wrapping each instance of "`semi-party.x`" in a program which provides provides inputs/outputs to the party. Since MP-SPDZ povides a universal interface our implementation is agnostic with regards to the underlying MPC implementation: any reactive MPC in MP-SPDZ which allows computation over $\mathbb{F}_{2^{16}+1}$ could be swapped in with ease.

2. *Linear Dependence on the Number of Parties.* A batched semi-honest version of CDN [CDN01] where we instantiate the linearly homomorphic encryption using the same ring LWE parameters described above. We implement this ourselves again using the Lattigo (more information below) library for the RLWE components.

CDN Implementation. We implement a semi-honest batched version of CDN, instantiating the linearly homomorphic encryption using the same parameters described above (the same as the OIP). To reduce the overhead (computational/communication) induced by the homomorphic encryption we execute multiplications in batches of 2^{12} (the dimension of the ring used for RLWE), by packing 2^{12} independent shares (over `0x10001`) into a single ciphertext and execute the CDN multiplication protocol on these in parallel. The decryption threshold is the full set of parties. The CDN implement is included in the same repository. To the best of our knowledge, this is the first known implementation of CDN.

Instantiating OIP and Ring LWE Parameters. In our implementation, we use an optimized version of OIP. We observe that the $O(n^2)$ overhead incurred from the use of pairwise-OIPs can be driven down to $O(n)$, if instead of a regular linearly homomorphic encryption, we use a threshold linearly homomorphic encryption (TLHE). TLHE are linearly homomorphic encryptions that comprise of a single public-key and where each party holds a "share" of the secret key. This share of the secret key can be used by the parties to decrypt to a share of the plaintext. As shown in [CHI+20], the keys for RLWE based threshold linearly homomorphic encryption can be setup very efficiently by the parties in a couple of rounds. At a high level, this observation allows us to reuse the sender and receiver messages of each party across multiple OIP instantiations and as a result, overall, each party only needs to send one receiver message and one sender message.

Recall that in our semi-honest protocol, the receiver and sender messages in all OIP instances are computed using the same shares of the index associated with

the active branch and the masks. Each party can compute its receiver message by encrypting its shares of $b_1, \ldots, b_k$. Similarly, for the sender message, each party can compute an inner-product of these encryptions received from all parties and its shares of the permuted masks. Finally, all parties can add all the sender messages (which are also ciphertexts) received from all parties. This gives them an encryption of the permuted masks for the active branch. Now each party can run threshold decryption using its share of the secret-key to obtain a sharing of the resulting inner-product.

We use BFV [FV12] over a cyclotomic ring of index 2^{13} and dimension 2^{12}, i.e. $R[X]/(X^{2^{12}}+1)$ where: $Q_1 := \texttt{0x7ffffec001}, Q_2 := \texttt{0x8000016001}, P := \texttt{0x40002001}$, $N := Q_1 Q_2 P, R := \mathbb{Z}/(N\mathbb{Z})$. This gives us a linearly homomorphic encryption scheme for vectors $\overrightarrow{v} \in (\mathbb{F}_{2^{16}+1})^{2^{12}}$, which additionally allows (full) threshold decryption. We use the `Lattigo` [lat21] library to implement all the RLWE components.

Benchmarking Platform. All benchmarks were run on a laptop with an Intel i7-11800H CPU (@ 2.3 GHz) and 64 GB of RAM. All networking is over the loopback interface and network latency was simulated using traffic control (`tc`) on Linux. We also do not restrict the bandwidth when comparing running times – note that this constitutes a relative "worst-case scenario" for our results: as our technique reduces communication, the relative performance gain for many branches would only increase by restricting bandwidth.

How The Branches Were Generated. During our benchmark each branch contained 2^{16} uniformly random gates: each gate is a multiplication/addition gate with probability $1/2$. We benchmark using "layered circuits", meaning each level contains 2^{12} gates which can be evaluated in parallel (to reduce the number of rounds). Subject to the layering constraint, the wiring is otherwise random: the inputs to each gate are sampled uniformly at random from all previous outputs (not just those in the last layer). We believe this distribution over circuits form a realistic benchmark for the expected performance across many real-world applications.

Averaging. We run all benchmarks 10 times and take the average.

Comparison of Communication Complexity

In Figure 6.8 and Figure 6.9, we compare the communication complexity of our technique to the naïve baseline solution of evaluating each branch in parallel using the underlying MPC. For the baseline solution we do not consider the additional overhead of multiplexing the output, i.e., selecting the output of the active branch.

We observe that our technique improves communication over the baseline for both CDN and MASCOT with 3 parties when the number of branches is ≥ 8. For less than 8 branches the communication overhead of the RLWE-based OIP and the need to evaluate universal gates (requiring the base-MPC to compute 3 multiplications) outweighs the communication saving of only executing the active branch. Upon reflection 8 branches is about the lowest number of branches we could hope to see

savings for: recall that each branch contains $\approx 2^{15}$ multiplications[3], therefore the parallel execution of 6 branches requires the same number of multiplications as that of the 2^{16} universal gates used in our technique. As expected we also observe that the communication of our technique remains (nearly[4]) constant for any number of branches.

Lastly we fix the number of branches to 16 and plot (in Figure 6.10) the communication complexity of our technique for a varying number of parties, as expected the communication of our compiler applied to MASCOT increases quadratically, while our technique preserves the linearly increasing communication of CDN; constant per-party communication (and computation).

Comparison of Running Time

From Figure 6.8 and Figure 6.9, we observe that for sufficiently many branches our technique also reduces running time over the baseline for both CDN and semi-honest MASCOT. This is also expected: after the relatively high constant overhead of our technique, the marginal cost of adding another branch (of length ℓ) is that of: (1) $O(\ell)$ linear operations in the underlying MPC. (2) $O(\ell)$ $\langle ciphertext \rangle \times \langle plaintext \rangle$ operations in the RLWE based homomorphic encryption scheme. (3) $O(\ell)$ $\langle ciphertext \rangle + \langle ciphertext \rangle$ operations in the RLWE based homomorphic encryption scheme.

The first one introduces a very small cost (essentially that of reading the branch), the second is dominated by the cost of doing a number theoretic transform (NTT) on the plaintext (the players local share), which again is essentially that of computing a small constant number of fixed-size FFTs. We note that the NTTs are computed on random shares and could be relegated to a pre-computation phase. The final ciphertext/ciphertext addition is just a constant number of entry-wise additions of vectors in a small prime field – the cost of which is miniscule. Looking at Figure 6.8 and Figure 6.9 we observe that this marginal computational cost (of doing NTTs) has a higher influence when the network latency is low and quickly becomes insignificant as the latency increases.

[3] Since the type of each gate in each branch is sampled uniformly at random.

[4] It grows slightly, since the unary representation of the selection wire must be shared/computed. However the computation of the branch completely dominates the communication.

Preprocessing Phase of the Maliciously Secure Protocol

The protocol is described in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model. Parties have shares of the inputs to the branches, i.e., $[x_1],\ldots,[x_\ell]$ and shares of a unary representation of the active branch, i.e., $[b_1],\ldots,[b_k]$. Let $\delta = \kappa/0.311$, where κ is the statistical security parameter.

Pre-processing Phase:

1. **Sample masks:** For each wire $w \in [W]$, parties invoke $(rand, \mathsf{mask}_w^1),\ldots,(rand,\mathsf{mask}_w^\delta)$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain shares $[\mathsf{mask}_w^1],\ldots,[\mathsf{mask}_w^\delta]$ respectively. For each $q \in [\delta]$, $m \in [k]$, let $[\overrightarrow{\mathsf{mask}^q_{\pi_m}}] = [\mathsf{mask}^q_{\pi_m(1)}]\|\ldots\|[\mathsf{mask}^q_{\pi_m(W)}]$.

2. **Shares of zeros:** For each $q \in [\delta]$, $w \in [W]$ and $i \in [n]$, the parties invoke $(sharezero, X_{w,i})$ in $\mathscr{F}_{\mathsf{mpc}}$ to get shares $[X^q_{w,i}]$, where $X^q_{w,i} = 0$. For each $i \in [n]$, let $[\overrightarrow{X^q_i}] = [X^q_{1,i}]\|\ldots\|[X^q_{k,i}]$.

3. **Pairwise OIP:** Each pair of parties P_R and P_S ($\forall \mathsf{R},\mathsf{S} \in [n]$) engage in two-message (malicious receiver) OIPs as follows, where P_R and P_S act as the receiver and sender resp.:

   - **Receiver:** P_R compute $(\rho, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}}) \leftarrow \mathsf{OIP}_\mathsf{R}(1^\lambda, [b_1]_\mathsf{R},\ldots,[b_k]_\mathsf{R})$, send msg_R to P_S.
   - **Sender:** For each $q \in [\delta]$, P_S samples random values $r^q_0,\ldots,r^q_k \in \mathbb{F}^{k+1}$ and does the following:
     - Computes:

$$\mathsf{msg}^q_{\mathsf{S}\to\mathsf{R}} \leftarrow \mathsf{OIP}_\mathsf{S}(1^\lambda, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}}, [\overrightarrow{X^q_\mathsf{R}}]_\mathsf{S}\|r^q_0, [\overrightarrow{\mathsf{mask}^q_{\pi_1}}]_\mathsf{S}\|r^q_1,\ldots,[\overrightarrow{\mathsf{mask}^q_{\pi_k}}]_\mathsf{S}\|r^q_k;\rho^{\mathsf{S},q}_{\mathsf{R},\mathsf{S}})$$

$$c^q_{\mathsf{R},\mathsf{S}} \leftarrow \mathsf{Commit}([\overrightarrow{X^q_\mathsf{R}}]_\mathsf{S}\|r^q_0, [\overrightarrow{\mathsf{mask}^q_{\pi_1}}]_\mathsf{S}\|r^q_1,\ldots,[\overrightarrow{\mathsf{mask}^q_{\pi_k}}]_\mathsf{S}\|r^q_k,\rho^{\mathsf{S},q}_{\mathsf{R},\mathsf{S}};\rho^{c,q}_{\mathsf{R},\mathsf{S}})$$

     - It sends $\mathsf{msg}^q_{\mathsf{S}\to\mathsf{R}}, c^q_{\mathsf{R},\mathsf{S}}$ to P_R.
   - **Output:** For each $q \in [\delta]$, P_R computes $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}}\|\mathsf{mac}^q_{\mathsf{R},\mathsf{S}} \leftarrow \mathsf{OIP}_{\mathsf{out}}(\rho, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}}, \mathsf{msg}_{\mathsf{S}\to\mathsf{R}})$.
   - **Receiver Consistency check:** For each $q \in [\delta]$, P_S invokes $(initinp, r^q_0, P_\mathsf{S}).\ldots,(initinp, r^q_k, P_\mathsf{S})$ and P_R invokes $(initinp, \mathsf{mac}^q_{\mathsf{R},\mathsf{S}}, P_\mathsf{R})$ in $\mathscr{F}_{\mathsf{mpc}}$. The parties then collectively invoke $(func, \mathscr{F}^q_{\mathsf{mac}}, r^q_0, r^q_1,\ldots,r^q_k,[b_1]_\mathsf{R},\ldots,[b_k]_\mathsf{R},\mathsf{out}^q_{\mathsf{mac}})$, where $\mathscr{F}^q_{\mathsf{mac}}(r^q_0,r^q_1,\ldots,r^q_k,[b_1]_\mathsf{R},\ldots,[b_k]_\mathsf{R}) = r^q_0 + \sum_{m\in[k]}[b_m]_\mathsf{R} r^q_m$. Finally, the parties invoke $(checkzero, \mathsf{out}^q_{\mathsf{mac}}, \mathsf{mac}^q_{\mathsf{R},\mathsf{S}})$ to check if $\mathsf{out}^q_{\mathsf{mac}} \stackrel{?}{=} \mathsf{mac}^q_{\mathsf{R},\mathsf{S}}$.

4. **Sender Consistency Check:** The parties use $\mathscr{F}_{\mathsf{mpc}}$ to sample a random subset $Z \subset [\delta]$ of size $\delta/2$ and then proceed as follows:

   - For each $q \in Z$, they invoke $(outshare, \mathsf{mask}^q_{\pi_1}),\ldots,(outshare,\mathsf{mask}^q_{\pi_k})$ and for each $i \in [n], w \in [W]$, they invoke $(outshare, X^q_{w,i})$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain all the shares of $\mathsf{mask}^q_{\pi_1}\ldots,\mathsf{mask}^q_{\pi_k}$ and $X^q_{w,i}$.
   - For each $q \in Z$, each pair of parties P_R and P_S ($\forall \mathsf{R},\mathsf{S} \in [n]$) do the following:
     a) P_S sends $r^q_0, r^q_1,\ldots,r^q_k,\rho^{\mathsf{S},q}_{\mathsf{R},\mathsf{S}},\rho^{c,q}_{\mathsf{R},\mathsf{S}}$ to P_R.
     b) P_R checks:

$$\mathsf{msg}^q_{\mathsf{S}\to\mathsf{R}} \stackrel{?}{=} \mathsf{OIP}_\mathsf{S}(1^\lambda, \mathsf{msg}_{\mathsf{R}\to\mathsf{S}}, [\overrightarrow{X^q_\mathsf{R}}]_\mathsf{S}\|r^q_0, [\overrightarrow{\mathsf{mask}^q_{\pi_1}}]_\mathsf{S}\|r^q_1,\ldots,[\overrightarrow{\mathsf{mask}^q_{\pi_k}}]_\mathsf{S}\|r^q_k;\rho^{\mathsf{S},q}_{\mathsf{R},\mathsf{S}})$$

$$c^q_{\mathsf{R},\mathsf{S}} \stackrel{?}{=} \mathsf{Commit}([\overrightarrow{X^q_\mathsf{R}}]_\mathsf{S}\|r^q_0, [\overrightarrow{\mathsf{mask}^q_{\pi_1}}]_\mathsf{S}\|r^q_1,\ldots,[\overrightarrow{\mathsf{mask}^q_{\pi_k}}]_\mathsf{S}\|r^q_k,\rho^{\mathsf{S},q}_{\mathsf{R},\mathsf{S}};\rho^{c,q}_{\mathsf{R},\mathsf{S}})$$

5. **Δ values:** If all the above checks succeed, then for each $q \in [\delta]\setminus Z$, each party P_i (for $i \in [n]$) computes $[\overrightarrow{\Delta^j}]_i = \sum_{j\in[n]} \overrightarrow{\mathsf{share}^q_{j,i}}$, where $[\overrightarrow{\Delta^q}] = [\Delta^q_1]\|\ldots\|[\Delta^q_W]$.

Figure 6.3: Pre-processing Phase of the Maliciously Secure Compiler

Online Phase of the Maliciously Secure Protocol

Online Phase : For each $q \in [\delta] \setminus Z$, parties compute the following:

1. **Inputs:** For each input wire $i \in [\ell]$, parties compute $[u_i^q] = [x_i] + [\mathsf{mask}_i]$ and invoke (out, u_i^q) in $\mathscr{F}_{\mathsf{mpc}}$ to obtain u_i^q in the clear.
2. **Circuit Evaluation:** For each gate $g \in [G]$, let $\mathsf{left} = \ell + 2g - 1$ and $\mathsf{right} = \ell + 2g$ be the incomining wire labels of its input wires. Let $\mathsf{type}_{m,g}$ be the gate type for gate g in C_m ($\forall m \in [k]$), where $\mathsf{type}_{m,g} = 0$ denotes an addition gate and $\mathsf{type}_{m,g} = 1$ denotes a multiplication gate.
   a) For $w \in \{\mathsf{left}, \mathsf{right}\}$, compute $[y_w^q] = \sum_{m=1}^{k} \left(u^q_{\pi_m(w)} \cdot [b_m] \right) + [\Delta_w^q]$.
   b) Compute $[\mathsf{type}_g] = \sum_{m=1}^{k} \left(\mathsf{type}_{m,g} \cdot [b_m] \right)$
   c) Compute $[z_g^q] = (1 - [\mathsf{type}_g])([y^q_{\mathsf{left}}] + [y^q_{\mathsf{right}}]) + [\mathsf{type}_g]([y^q_{\mathsf{left}}] \cdot [y^q_{\mathsf{right}}])$.
   d) Compute $[u^q_{\ell+g}] = [z_g^q] + [\mathsf{mask}_{\ell+g}]$ and invoke $(out, u^q_{\ell+g})$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain $u^q_{\ell+g}$ in the clear.
3. For each output gate g, compute $[z_g^q] = \sum_{m=1}^{k} \left(u^q_{\pi_m(w)} \cdot [b_m] \right) + [\Delta_w^q]$.

Output For each output gate g, the parties invoke $(func, \mathsf{maj}, z_g^1, \ldots, z_g^\delta, z_g)$ to get shares $[z_g]$, where the maj is the majority function.

Figure 6.4: Online Phase of the Maliciously Secure Compiler

Garbling Phase of the Constant Round Semi-Honest Protocol

The protocol is described in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model which computes over $\mathscr{R}_p$. The parties have shares of a unary representation of the active branch, i.e., $[b_1],\ldots,[b_k]$. For each gate $g \in [G]$, let $\mathsf{left}_g = \ell + 2g - 1$ and $\mathsf{right}_g = \ell + 2g$ be the incoming wire labels of its input wires and let $\mathsf{out}_g = \ell + g$ be the outgoing wire.

1. **Sample masks:** For each input and gate $g \in [\ell + G]$, the parties invoke $(\mathit{randbit}, \gamma_g)$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain shares $[\gamma_g]$. For each branch $m \in [k]$, let $[\overrightarrow{\gamma_{\pi_m}}] = [\gamma_{\pi_m(1)}] \| \ldots \| [\gamma_{\pi_m(W)}]$.
2. **Sample keys:** For each $g \in [\ell + G]$, and $j \in \{0,1\}$ each party P_i (for $i \in [n]$) locally samples its share $[k_g^j]_i \leftarrow \chi^N$ and sets the last coordinate of its share to 0.
3. **Compute LWE expansions:** For each $u, v \in \{0,1\}$, $g \in [G]$ each party P_i (for $i \in [n]$) locally samples $\delta_{m,g}^{u,v,i} \leftarrow \chi^N$ and computes $[\psi_{m,g}^{u,v}]_i = A_g^{u,v} \cdot ([k_{\pi_m(\mathsf{left}_g)}]_i + [k_{\pi_m(\mathsf{right}_g)}]_i) + \delta_{m,g}^{u,v,i}$. Let $[\overrightarrow{\psi_m}] = [\psi_{m,1}^{0,0}] \| [\psi_{m,1}^{0,1}] \| [\psi_{m,1}^{1,0}] \| [\psi_{m,1}^{1,1}] \| \ldots \| [\psi_{m,G}^{0,0}] \| [\psi_{m,G}^{0,1}] \| [\psi_{m,G}^{1,0}] \| [\psi_{m,G}^{1,1}]$.
4. **Shares of zero:** For each $i \in [n]$ and $j \in [W + 4G]$, the parties invoke $(\mathit{sharezero}, X_{j,i})$ in $\mathscr{F}_{\mathsf{mpc}}$ to get shares $[X_{j,i}]$, where $X_{j,i} = 0$. For each $i \in [n]$, let $[\overrightarrow{X_i}] = [X_{1,i}] \| \ldots \| [X_{W+4G,i}]$.
5. **Pairwise OIP:** Each pair of parties P_{R} and P_{S} ($\forall \mathsf{R}, \mathsf{S} \in [n]$) engage in a two-message semi-honest OIP as follows, where P_{R} acts as the receiver and P_{S} acts as the sender:
   - **Receiver:** P_{R} computes $(\rho, \mathsf{msg}_{\mathsf{R}}) \leftarrow \mathsf{OIP}_{\mathsf{R}}(1^\lambda, [b_1]_{\mathsf{R}}, \ldots, [b_k]_{\mathsf{R}})$ and sends $\mathsf{msg}_{\mathsf{R}}$ to P_{S}.
   - **Sender:** For each $m \in [1,k]$ let $[\overrightarrow{x_m}] = [\overrightarrow{\gamma_{\pi_m}}] \| [\overrightarrow{\psi_m}]$. P_{S} computes $\mathsf{msg}_{\mathsf{S}} \leftarrow \mathsf{OIP}_{\mathsf{S}}(1^\lambda, \mathsf{msg}_{\mathsf{R}}, [\overrightarrow{X_{\mathsf{R}}}]_{\mathsf{S}}, [\overrightarrow{x_1}]_{\mathsf{S}}, \ldots, [\overrightarrow{x_k}]_{\mathsf{S}})$ and sends $\mathsf{msg}_{\mathsf{S}}$ to P_{R}.
   - **Output:** P_{R} computes $\overrightarrow{\mathsf{share}_{\mathsf{R},\mathsf{S}}} \leftarrow \mathsf{OIP}_{\mathsf{out}}(\rho, \mathsf{msg}_{\mathsf{R}}, \mathsf{msg}_{\mathsf{S}})$.

   For each $i \in [n]$, P_i computes $[\overrightarrow{\Gamma}] \| [\overrightarrow{\Psi}] = \sum_{j \in [n]} \overrightarrow{\mathsf{share}_{j,i}}$ where $\overrightarrow{\Gamma} = \Gamma_1 \| \ldots \| \Gamma_W$ and $\overrightarrow{\Psi} = \Psi_1^{0,0} \| \Psi_1^{0,1} \| \Psi_1^{1,0} \| \Psi_1^{1,1} \| \ldots \| \Psi_G^{0,0} \| \Psi_G^{0,1} \| \Psi_G^{1,0} \| \Psi_G^{1,1}$.
6. **Garble active branch:** Let $\mathsf{type}_{m,g}$ be the gate type for gate g in C_m ($\forall m \in [k]$), where $\mathsf{type}_{m,g} = 0$ denotes an XOR gate and $\mathsf{type}_{m,g} = 1$ denotes an AND gate. Parties do the following for each $g \in [G]$
   a) Compute $[\mathsf{type}_g] = \sum_{m=1}^k \left(\mathsf{type}_{m,g} \cdot [b_m]\right)$.
   b) For each $u, v \in \{0,1\}$ let $e_{u,v,g}^{\mathrm{xor}} = u \oplus \Gamma_{\mathsf{left}_g} \oplus v \oplus \Gamma_{\mathsf{right}_g} \oplus \gamma_{\mathsf{out}_g}$, $e_{u,v,g}^{\mathrm{and}} = ((u \oplus \Gamma_{\mathsf{left}_g}) \wedge (v \oplus \Gamma_{\mathsf{right}_g})) \oplus \gamma_{\mathsf{out}_g}$, $e_g^{u,v} = \mathsf{type}_g(e_{u,v,g}^{\mathrm{and}} - e_{u,v,g}^{\mathrm{xor}}) + e_{u,v,g}^{\mathrm{xor}}$ and $K_g^{u,v} = e_g^{u,v}(k_{\mathsf{out}_g}^1 - k_{\mathsf{out}_g}^0) + k_{\mathsf{out}_g}^0$. For each $u, v \in \{0,1\}$, compute $[K_g^{u,v} \| e_g^{u,v}]$ using $\mathscr{F}_{\mathsf{mpc}}$.
   c) For each $u, v \in \{0,1\}$ compute $[C_g^{u,v}] = [\Psi_g^{u,v}] + \lceil \sqrt{p} \rceil [K_g^{u,v} \| e_g^{u,v}]$.

Figure 6.5: Garbling phase of the constant round (semi-honest) protocol

Evaluation Phase of the Constant Round Semi-Honest Protocol

The protocol is described in the $\mathscr{F}_{\mathsf{mpc}}$-hybrid model. The parties have shares of the inputs to the branches, i.e., $[x_1],\ldots,[x_\ell]$ and shares of a unary representation of the active branch, i.e., $[b_1],\ldots,[b_k]$.

1. For each input wire $w \in [\ell]$ parties compute $[\beta_w] = [x_w] \oplus [\gamma_w]$ and invoke $(out, [\beta_w])$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain β_w. For each $w \in [\ell]$, let $\beta_{1,w} = \ldots = \beta_{k,w} = \beta_w$.
2. For each input wire $w \in [\ell]$ parties invoke $(out, [k_w^{\beta_w}])$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain $k_w^{\beta_w}$. For each $w \in [\ell]$, let $K_{1,w}^{\beta_w} = \ldots = K_{k,w}^{\beta_w} = k_w^{\beta_w}$.
3. For each $u,v \in \{0,1\}$ and $g \in [G]$ parties invoke $(out, [C_g^{u,v}])$ in $\mathscr{F}_{\mathsf{mpc}}$ to obtain $C_g^{u,v}$.
4. For each $m \in [k]$ and $g \in [G]$, parties compute $C_g^{u,v} - A_g^{u,v} \cdot \left(K^u_{m,\pi_m(\mathsf{left}_g)} + K^v_{m,\pi_m(\mathsf{right}_g)}\right)$, where $u = \beta_{m,\pi_m(\mathsf{left}_g)}$ and $v = \beta_{m,\pi_m(\mathsf{right}_g)}$, and divide it by $\lceil\sqrt{p}\rceil$ to remove the error and recover $K_{m,\mathsf{out}_g}^{\beta_{m,\mathsf{out}_g}} \| \beta_{m,\mathsf{out}_g}$.
5. For each output gate g, parties compute $[z_g] = \sum_{m=1}^{k} \beta_{m,\mathsf{out}_g}[b_m] \oplus [\gamma_g]$ using $\mathscr{F}_{\mathsf{mpc}}$.

Figure 6.6: Evaluation phase of the constant round (semi-honest) protocol

Two-message Semi-honest OIP from Linearly Homomorphic encryption.

Let $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a linearly homomorphic encryption. The parties proceed as follows:

Receiver: The receiver with inputs $b_1, \ldots, b_k \in \mathscr{M}$, proceeds as follows:

- It computes $(\mathrm{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$.
- For each $i \in [k]$, it computes $c_i \leftarrow \mathsf{Enc}_(\mathrm{pk}, \mathsf{b_i})$.
- It sends $\mathsf{msg_R} = (\mathrm{pk}, c_1, \ldots, c_k)$ to the sender.

Sender: The sender with inputs $\overrightarrow{m_0}, \ldots, \overrightarrow{m_k} \in \mathscr{M}^{m \times (k+1)}$, parses $\mathsf{msg_R} = (\mathrm{pk}, c_1, \ldots, c_k)$ and computes the following for each $j \in [m]$,:

- It computes $c_0 \leftarrow \mathsf{Enc}_(\mathrm{pk}, \mathsf{m_0[j]})$.
- It sets L_j to be the linear function $L_j(x_1, \ldots, x_k) = \sum_{i \in [k]} x_i \cdot \overrightarrow{m}_i[j]$, where $\overrightarrow{m_i}[j]$ is the j^{th} element in the vector $\overrightarrow{m_i}$.
- It computes $c_{L_j} \leftarrow \mathsf{Eval}(\mathrm{pk}, L_j, c_1, \ldots, c_k) + c_0$.

It sends $\mathsf{msg_S} = (c_{L_1}, \ldots, c_{L_m})$ to the receiver.

Output: The receiver parses $\mathsf{msg_S} = (c_{L_1}, \ldots, c_{L_m})$ and for each $j \in [m]$, it computes $\overrightarrow{x}[j] \leftarrow \mathsf{Dec}(\mathsf{sk}, c_{L_j})$ and outputs $\overrightarrow{x}$.

Figure 6.7: Two-message semi-honest OIP from linearly homomorphic encryption.

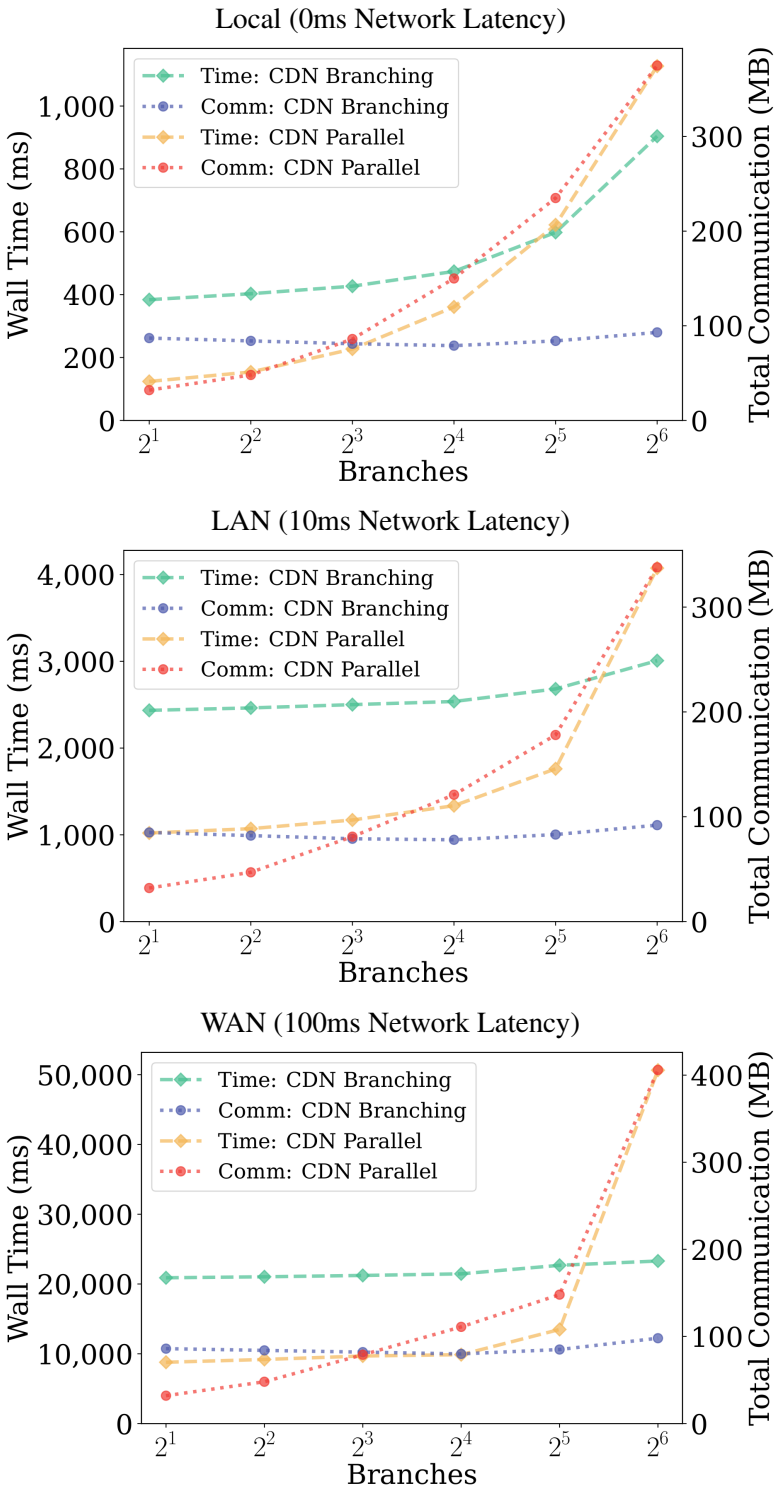

Figure 6.8: Running time of Branching MPC with CDN.

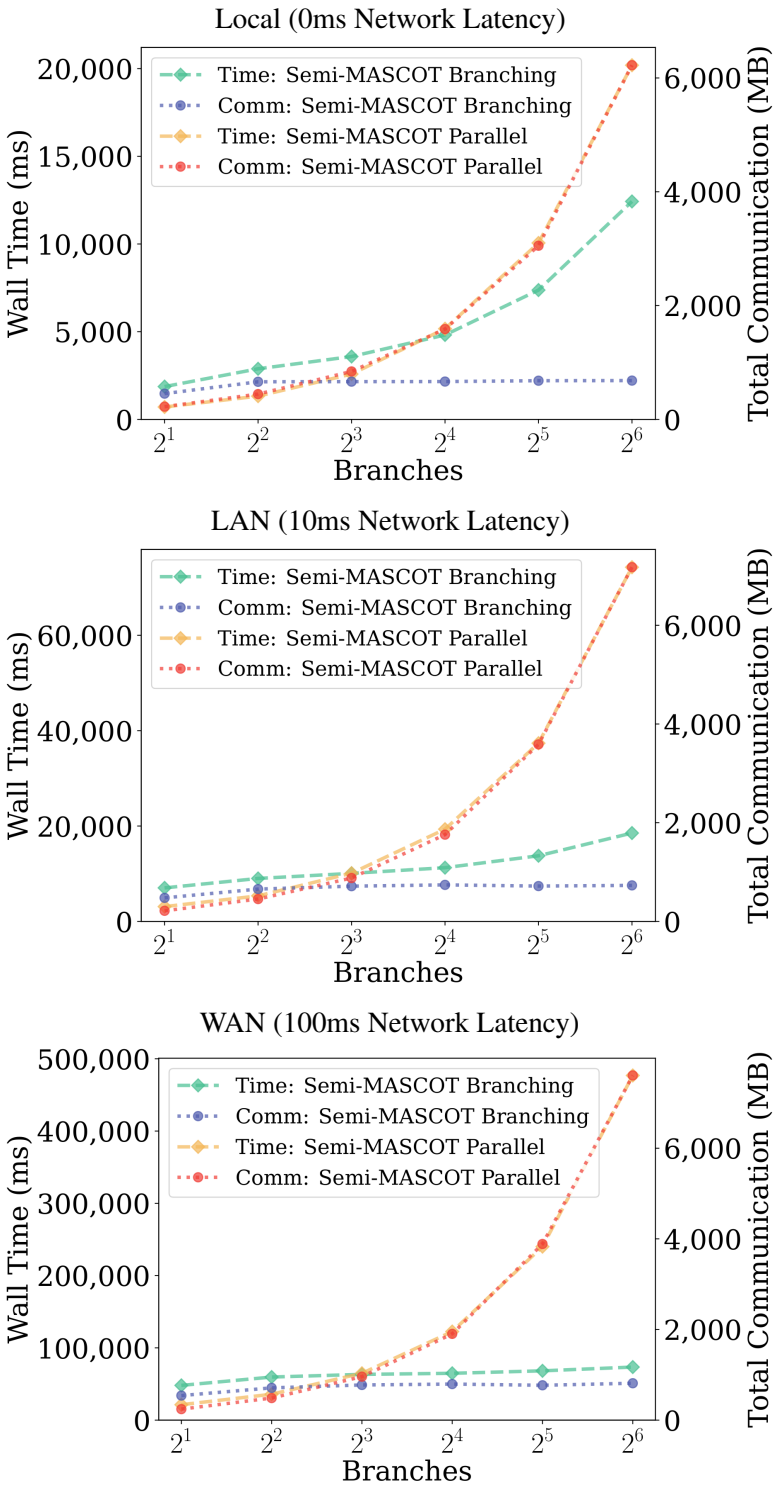

Figure 6.9: Running time of Branching MPC with Semi-Honest MASCOT.

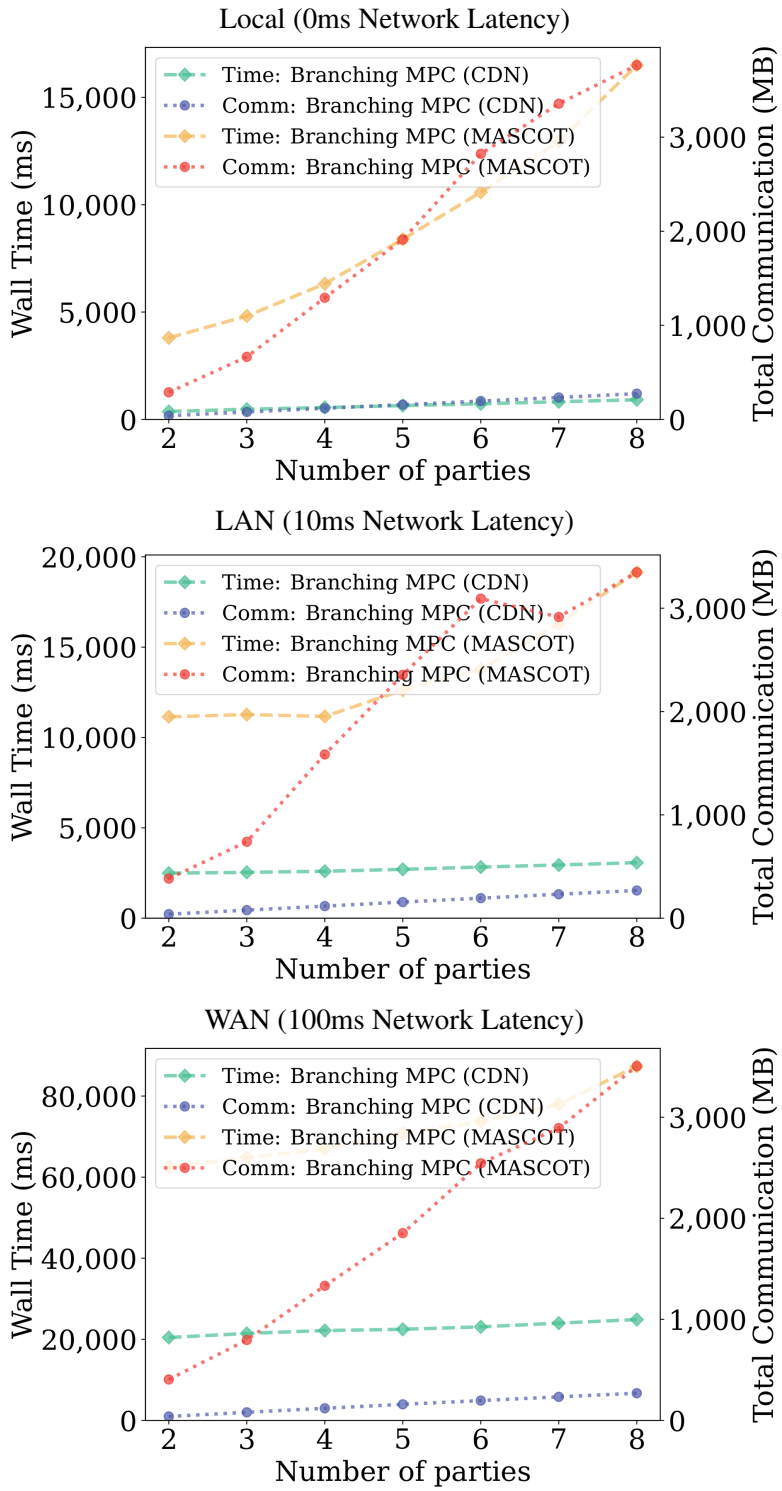

Figure 6.10: Running time of Branching MPC for Different Number of Parties.

Bibliography

[ABFV22] Gennaro Avitabile, Vincenzo Botta, Daniele Friolo, and Ivan Visconti. Efficient proofs of knowledge for threshold relations. Cryptology ePrint Archive, Report 2022/746, 2022. https://eprint.iacr.org/2022/746. 49, 77, 80

[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. 39, 62, 110

[ACD+16] Masayuki Abe, Melissa Chase, Bernardo David, Markulf Kohlweiss, Ryo Nishimaki, and Miyako Ohkubo. Constant-size structure-preserving signatures: Generic constructions and simple assumptions. *Journal of Cryptology*, 29(4):833–878, October 2016. 105

[ACF20] Thomas Attema, Ronald Cramer, and Serge Fehr. Compressing proofs of k-out-of-n partial knowledge. 2020. https://eprint.iacr.org/2020/753. 37, 39

[ACK21] Thomas Attema, Ronald Cramer, and Lisa Kohl. A compressed Σ-protocol theory for lattices. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 549–579, Virtual Event, August 2021. Springer, Heidelberg. 93

[AD18] Tomer Ashur and Siemen Dhooghe. MARVELlous: a STARK-friendly family of cryptographic primitives. Cryptology ePrint Archive, Report 2018/1098, 2018. https://eprint.iacr.org/2018/1098. 19

[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. 37, 38, 45, 46, 67, 68, 72, 73, 76, 88, 89

[AJ18] Kurt M. Alonso and Jordi Herrera Joancomartí. Monero - privacy in the blockchain. Cryptology ePrint Archive, Report 2018/535, 2018. https://eprint.iacr.org/2018/535. 131

[AOS02] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. 1-out-of-n signatures from a variety of keys. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 415–432. Springer, Heidelberg, December 2002. 16, 35, 36, 39, 41, 57

[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. 35, 110, 111

[BBC+19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Heidelberg, August 2019. 15

[BBD+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 581–615. Springer, Heidelberg, August 2023. 17

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019. 132

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046. 19

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019. 19

[BC23] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. https://eprint.iacr.org/2023/620. 19

[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. 37

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 111–120. ACM Press, June 2013. 19

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 301–315, 2017. 106

[BCF+21] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 393–414, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg. 104

[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. 18, 36

[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. 18, 35

[BCG+18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 595–626. Springer, Heidelberg, December 2018. 18

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018. 17

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 681–710, Virtual Event, August 2021. Springer, Heidelberg. 19

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 1–18. Springer, Heidelberg, November 2020. 19

[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. 35

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014. 19

[BCTV14b] 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. 35, 36

[Bd94] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994. 18, 106

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006. 80

[BFB21] Guillaume Ballet, Dankrad Feist, and Vitalik Buterin. Verkle tree EIP, 2021. `https://notes.ethereum.org/@vbuterin/verkle_tree_eip`. 107

[BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 390–420. Springer, Heidelberg, August 1993. 10, 11

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. `https://eprint.iacr.org/2019/1021`. 19

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 509–539. Springer, Heidelberg, August 2016. 21, 133

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 663–693. Springer, Heidelberg, October 2021. 133

[BGL20] Eli Ben-Sasson, Lior Goldberg, and David Levit. STARK friendly hash – survey and recommendation. Cryptology ePrint Archive, Report 2020/948, 2020. `https://eprint.iacr.org/2020/948`. 107

[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. 63, 85, 133

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013. 46

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013. 144

[BLO17] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Efficient scalable constant-round MPC via garbled circuits. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 471–498. Springer, Heidelberg, December 2017. 143, 144, 145, 161, 162

[Blu87] Manuel Blum. How to prove a theorem so no one else can claim it. pages 1444–1451, 1987. 38, 58, 62, 82

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990. 135, 142

[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. 40

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Heidelberg. 17

[BPRS23] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: Souper fast three-party computation for ram programs. Cryptology ePrint Archive, Paper 2023/310, 2023. https://eprint.iacr.org/2023/310. 21

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. 13

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012. 27

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Heidelberg, August 1988. 133

[CD98] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 424–441. Springer, Heidelberg, August 1998. 61, 73

[CDE+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod 2^k for dishonest majority. Cryptology ePrint Archive, Report 2018/482, 2018. https://eprint.iacr.org/2018/482. 133, 135, 138, 144, 149

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Heidelberg, May 2001. 20, 27, 135, 136, 149, 167

[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. 16, 23, 35, 36, 39, 41, 58, 79, 96

[CFGG22] Dario Catalano, Dario Fiore, Rosario Gennaro, and Emanuele Giunta. On the impossibility of algebraic vector commitments in pairing-free groups. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 274–299. Springer, Heidelberg, November 2022. 106

[CFH+22] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 455–469. ACM Press, November 2022. 104

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019. 105

[CGG+23] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. Cryptology ePrint Archive, Paper 2023/902, 2023. `https://eprint.iacr.org/2023/902`. 16, 17

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *30th ACM STOC*, pages 209–218. ACM Press, May 1998. 13

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018. 133

[CHA21] Matteo Campanelli and Mathias Hall-Andersen. Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. Cryptology ePrint Archive, Report 2021/327, 2021. `https://ia.cr/2021/327`. 104, 105, 124

[CHA22a] Matteo Campanelli and Mathias Hall-Andersen. Curve trees: Practical and transparent zero-knowledge accumulators. Cryptology ePrint Archive, Report 2022/756, 2022. `https://eprint.iacr.org/2022/756`. 18

[CHA22b] Matteo Campanelli and Mathias Hall-Andersen. Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 652–666. ACM Press, May / June 2022. 18

[CHA22c] Matteo Campanelli and Mathias Hall-Andersen. Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 652–666, 2022. 124, 131

[CHI+20] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. Cryptology ePrint Archive, Report 2020/374, 2020. `https://eprint.iacr.org/2020/374`. 104, 167

[CHL+05] Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knowledge sets. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 422–439. Springer, Heidelberg, May 2005. 106

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020. 25

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002. 106

[CL15] Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from DDH. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 487–505. Springer, Heidelberg, April 2015. 137, 154, 166

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In Shafi Goldwasser, editor, *ITCS 2012*, pages 90–112. ACM, January 2012. 8

[Con22] Graeme Connell. Signal blog: Technology deep dive: Building a faster ORAM layer for enclaves, 2022. `https://signal.org/blog/building-faster-oram/`. 21

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020. 17, 19

[CP93] David Chaum and Torben P. Pedersen. Transferred cash grows in size. In Rainer A. Rueppel, editor, *EUROCRYPT'92*, volume 658 of *LNCS*, pages 390–407. Springer, Heidelberg, May 1993. 62, 73

[CP14] Nikolaos Triandopoulos Charalampos Papamanthou, Roberto Tamassia. U.S Patent. US9098725B2, Cryptographic accumulators for authenticated hash tables, 2014. https://patents.google.com/patent/US9098725B2/en. 107

[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. 39, 42

[CS97] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. 1997. 14, 109

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In Andrew Chi-Chih Yao, editor, *ICS 2010*, pages 310–331. Tsinghua University Press, January 2010. 19

[Dam92] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 445–456. Springer, Heidelberg, August 1992. 11, 12

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 378–394. Springer, Heidelberg, August 2005. 143

[DILO22] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 829–841. ACM Press, November 2022. 17

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-Point Zero Knowledge and Its Applications. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, volume 199

of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 17

[DJ01] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001. 137, 154, 166

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012. 133, 136, 138, 144, 149

[DT08] Ivan Damgård and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538, 2008. https://eprint.iacr.org/2008/538. 106

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, Heidelberg, January 2005. 127

[Eag22] Liam Eagen. Bulletproofs++. Cryptology ePrint Archive, Report 2022/510, 2022. https://eprint.iacr.org/2022/510. 18

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. 12

[FKM+16] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. Efficient unlinkable sanitizable signatures from signatures with re-randomizable keys. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 301–330. Springer, Heidelberg, March 2016. 125

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to mpc with preprocessing using ot. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 711–735, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. 135

[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`. 51

[FMMO19] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 649–678. Springer, Heidelberg, December 2019. 131

[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. 40

[FOSZ23] Brett Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. GigaDORAM: Breaking the billion address barrier. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3871–3888, Anaheim, CA, August 2023. USENIX Association. 21

[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. 36, 37, 57, 80, 117

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`. 27, 137, 154, 166, 168

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009. 133

[GGHAK22] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Stacking sigmas: A framework to compose Σ-protocols for disjunctions. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 458–487. Springer, Heidelberg, May / June 2022. 16, 17, 23

[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. 36

[GHAHJ22] Aarushi Goel, Mathias Hall-Andersen, Aditya Hegde, and Abhishek Jain. Secure multiparty computation with free branching. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 397–426. Springer, Heidelberg, May / June 2022. 20, 25, 26

[GHAK23] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. Dora: Processor expressiveness is (nearly) free in zero-knowledge for ram programs. Cryptology ePrint Archive, Paper 2023/1749, 2023. `https://eprint.iacr.org/2023/1749`. 17

[GHAKS23] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. Speed-stacking: Fast sublinear zero-knowledge proofs for disjunctions. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 347–378. Springer, Heidelberg, April 2023. 17

[Gil52] E. N. Gilbert. A comparison of signalling alphabets. *The Bell System Technical Journal*, 31(3):504–522, 1952. 7

[GK90] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. In *International Colloquium on Automata, Languages and Programming*, 1990. 9

[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. 39

[GK16] Shafi Goldwasser and Yael Tauman Kalai. Cryptographic assumptions: A position paper. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 505–522. Springer, Heidelberg, January 2016. 12

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008. 8

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021. 18, 19, 107, 130

[GLO+21] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021,*

Part II, volume 12826 of *LNCS*, pages 244–274, Virtual Event, August 2021. Springer, Heidelberg. 133, 138, 144

[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. 35

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. 9, 10

[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. 35

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987. 20, 133, 140

[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. 35

[GO94] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994. 68

[GO96a] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996. 21

[GO96b] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. 21

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987. 21

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004. 145

[GOP+16] Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set algebra. In Jung Hee Cheon and Tsuyoshi Takagi, editors,

ASIACRYPT 2016, Part II, volume 10032 of *LNCS*, pages 67–100. Springer, Heidelberg, December 2016. 106

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 275–304, Virtual Event, August 2021. Springer, Heidelberg. 133

[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. 38, 45, 61

[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. 36

[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. 35, 36

[GS20] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134, 2020. `https://eprint.iacr.org/2020/134`. 133

[GSY21] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 694–723. Springer, Heidelberg, October 2021. 133

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011. 12, 13

[GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020. `https://eprint.iacr.org/2020/315`. 18

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://eprint.iacr.org/2019/953`. 17, 18, 25, 107

[Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Report 2022/1530, 2022. `https://eprint.iacr.org/2022/1530`. 18

[HBHW21] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, version 2021.2.16 [nu5 proposal], 2021. 18, 104, 107

[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. 40, 134, 136, 137

[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. 16, 20, 35, 36, 37, 40, 41

[HK21] David Heath and Vladimir Kolesnikov. LogStack: Stacked garbling with $O(b \log b)$ computation. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 3–32. Springer, Heidelberg, October 2021. 20, 134, 136, 137

[HKO22] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. EpiGRAM: Practical garbled RAM. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 3–33. Springer, Heidelberg, May / June 2022. 21

[HKP20] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 3–30. Springer, Heidelberg, December 2020. 20, 134, 136, 137

[HKP21a] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Garbling, stacked and staggered - faster k-out-of-n garbled function evaluation. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 245–274. Springer, 2021. 134

[HKP21b] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Masked triples - amortizing multiplication triples across conditionals. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 319–348. Springer, Heidelberg, May 2021. 134, 136, 137

[HKRS20] Marco Holz, Ágnes Kiss, Deevashwer Rathee, and Thomas Schneider. Linear-complexity private function evaluation is practical. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 401–420. Springer, Heidelberg, September 2020. 20

[Hop20] Daira Hopwood, 2020. `https://github.com/zcash/pasta`. 109, 127

[HOSS18] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 86–117. Springer, Heidelberg, December 2018. 15, 133, 135, 138, 144, 149

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003. 17

[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. 15, 45, 48, 57, 62, 64, 85, 86

[IP07] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 575–594. Springer, Heidelberg, February 2007. 21

[JB23] Paul Gafni Jeremy Bruestle. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity, 2023. 18

[Jiv19] Aram Jivanyan. Lelantus: A new design for anonymous and confidential cryptocurrencies. Cryptology ePrint Archive, Paper 2019/373, 2019. `https://eprint.iacr.org/2019/373`. 131

[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. 35, 36, 40

[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. 39

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020. 167

[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. viii, 35, 38, 45, 46, 67, 72, 73, 76, 79, 85, 86, 87

[KM11] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 556–571. Springer, Heidelberg, December 2011. 20, 137

[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. 20, 36, 40, 66

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016. 15, 133, 135, 136, 138, 144, 149, 150, 167

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018. 135

[KS16] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 699–728. Springer, Heidelberg, May 2016. 16

[KS22] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758, 2022. https://eprint.iacr.org/2022/1758. 19

[KS23] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573. 19

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Heidelberg, August 2022. 19

[lat21] Lattigo v2.2.0. Online: `http://github.com/ldsec/lattigo`, July 2021. EPFL-LDS. 168

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`. 23

[LFKN90] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *31st FOCS*, pages 2–10. IEEE Computer Society Press, October 1990. 8

[Lip16] Helger Lipmaa. Prover-efficient commit-and-prove zero-knowledge SNARKs. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 185–206. Springer, Heidelberg, April 2016. 105

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2018. 21

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013. 21

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, Heidelberg, March 2011. 160

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010. 161

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, Heidelberg, May 2013. 161

[LRR+19] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. Omniring: Scaling private payments without trusted setup. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 31–48. ACM Press, November 2019. 104, 131

[Mau05] Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, December 2005. 13

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988. 24

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013. 18, 131

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30:1253–1298, 2000. 19

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, Heidelberg, August 1986. 108

[min23] Mina Protocol, 2023. `https://minaprotocol.com/`. 20

[MRK03] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *44th FOCS*, pages 80–91. IEEE Computer Society Press, October 2003. 106

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer, Heidelberg, May 2013. 20, 26, 137, 138

[MSK02] Shigeo Mitsunari, Ryuichi Sakai, and Masao Kasahara. A new traitor tracing. *IEICE Transactions*, E85-A(2):481–484, February 2002. 127

[MSS14] Payman Mohassel, Seyed Saeed Sadeghian, and Nigel P. Smart. Actively secure private function evaluation. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 486–505. Springer, Heidelberg, December 2014. 20, 137

[Nao03] Moni Naor. On cryptographic assumptions and challenges (invited talk). In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 96–109. Springer, Heidelberg, August 2003. 12

[Neu51] Von Neumann. Various techniques used in connection with random digits. *Notes by GE Forsythe*, pages 36–38, 1951. 3

[Ngu05] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 275–292. Springer, Heidelberg, February 2005. 106

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999. 144

[PLS23] Andrew Park, Wei-Kai Lin, and Elaine Shi. NanoGRAM: Garbled RAM with $\widetilde{O}(\log N)$ overhead. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 456–486. Springer, Heidelberg, April 2023. 21

[PTT08] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 437–448. ACM Press, October 2008. 106

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2008. 137, 154, 166

[R.57] Varshamov R. R. Estimate of the number of signals in error correcting codes. *Docklady Akad. Nauk, S.S.S.R.*, 117:739–741, 1957. 7

[rep20] Report on the security of stark-friendly hash functions (version 2.0), 2020. 107

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Heidelberg, August 2022. 17

[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. 36, 79

[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. 38, 45, 61, 79, 80

[se19] swisspost evoting. E-voting system 2019. `https://gitlab.com/swisspost-evoting/e-voting-system-2019`, 2019. 35

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020. 8

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. 63

[Sha90] Adi Shamir. IP=PSPACE. In *31st FOCS*, pages 11–15. IEEE Computer Society Press, October 1990. 8

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997. 13

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. `https://eprint.iacr.org/2023/552`. 18

[SvS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013. 21

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 71–89. Springer, Heidelberg, August 2013. 8

[Tha23] Justin Thaler. Proofs, arguments, and zero-knowledge (draft: 2023-07-18), 2023. `https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK`. 8

[The] The Coq Development Team. Coq. 8

[Val76] Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, page 196–203, New York, NY, USA, 1976. Association for Computing Machinery. 16

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 1–18. Springer, Heidelberg, March 2008. 18

[van90] Peter van EMDE BOAS. Chapter 1 - machine models and simulations. In JAN VAN LEEUWEN, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 1–66. Elsevier, Amsterdam, 1990. 12

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015. 21

[WJS+19] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. Stormy: Statistics in tor by measuring securely. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 615–632. ACM Press, November 2019. 133

[WK19] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. 2019. Available at `http://plfa.inf.ed.ac.uk/`. 8

[WSR+15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, February 2015. 18

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021. 17

[XZZ+19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, August 2019. 8

[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. 133, 142

[YHH+23] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Batchman and robin:

Batched and non-batched branching for interactive zk. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1452–1466, New York, NY, USA, 2023. Association for Computing Machinery. 17

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021. 15, 17

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020. 17

[Zav20] Greg Zaverucha. The picnic signature algorithm. Technical report, 2020. `https://github.com/microsoft/Picnic/raw/master/spec/spec-v3.0.pdf`. 35

[ZBK+22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3121–3134. ACM Press, November 2022. 18, 105, 106

[Zha22] Mark Zhandry. To label, or not to label (in generic groups). In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part III*, volume 13509 of *LNCS*, pages 66–96. Springer, Heidelberg, August 2022. 13

[zke23] ZK-EVM Circuits. Github Repository, 2023. 18

[ZZK22] Cong Zhang, Hong-Sheng Zhou, and Jonathan Katz. An analysis of the algebraic group model. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 310–322. Springer, Heidelberg, December 2022. 13