ON DEPLOYING SUCCINCT ZERO-KNOWLEDGE PROOFS

by

MADARS VIRZA

Master of Science, Massachusetts Institute of Technology (2014)
Bachelor of Science, University of Latvia (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

Author .... Signature redacted
Department of Electrical Engineering and Computer Science
August 31, 2017

Certified by .... Signature redacted
Ronald L. Rivest
Institute Professor
Thesis Supervisor

Accepted by .... Signature redacted
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

ON DEPLOYING SUCCINCT ZERO-KNOWLEDGE PROOFS

by

MADARS VIRZA

Submitted to the Department of Electrical Engineering and Computer Science on August 31, 2017, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Zero-knowledge proofs, introduced by Goldwasser, Micali, and Rackoff, are a fundamental building block in theoretical cryptography with numerous applications. Still, the impact of zero-knowledge proofs for building secure systems in practice has been modest at best. Part of this can be explained by the economics of deploying new technology in the wild: often introducing a trusted third party in lieu of a proof system achieves users' security goals with lower anticipated cost.

The goal of this thesis is to lower the cost of using zero-knowledge proofs in real-world systems. This cost has two major components: the cost incurred by the proof system itself, and the price paid to instantiate the security model the proof system relies on. Working with my collaborators, I have contributed to reducing both of these costs:

- **Cost of the security model.** For many practical scenarios it is crucial that proofs be non-interactive and succinct. In the standard model, non-interactive zero-knowledge (NIZK) proofs do not exist for languages outside BPP (even with just computational soundness). However, if the security model includes a trusted party, available for a one-time setup phase, then NIZKs exist for all languages in NP. Soundness of the NIZK depends on this trusted setup: if public parameters are not correctly generated, or if the trusted party's secret internal randomness is revealed, an attacker could convince the verifier of false NP statements without being detected. We show how public parameters for a class of NIZKs can be generated by a concretely-efficient multi-party protocol, such that if at least one of the parties is honest, then the result is secure and can be subsequently used for generating and verifying numerous proofs without any further trust.
- **Cost of the proof system.** We have designed and built an open-source cryptographic library, called **libsnark**, that provides efficient implementations of state-of-the-art zero-knowledge proof constructions. Our library is the fastest and most comprehensive suite of zero-knowledge proofs currently available.

Working in tandem, these contributions have achieved industrial impact, and are the main efficiency enablers for Zerocash, a privacy-preserving payment system.

Thesis Supervisor: Ronald L. Rivest

Title: Institute Professor

Contents

Previously Published Material

The zk-SNARK definitions in Chapter 1 follow [BCG$^+$14]. Chapter 2 describes a software library, called **libsnark** [SCI], that has been built to achieve results of [BCG$^+$13a, BCTV14a, BCTV14c, CTV15, BCG$^+$14]; these papers also report on certain aspects of **libsnark**'s design and we rely on parts of this exposition. Chapter 3 revises and extends a previous publication [BCG$^+$15]: E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza "Secure sampling of public parameters for succinct zero knowledge proofs." In *proceedings of the 2015 IEEE Symposium on Security and Privacy* (S&P '15).

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Ron Rivest, for his supervision, guidance, and insights over the past six academic years. I'm very grateful for Ron's excellent advice, encouragement, and everything I have learned from him. I am honored that Eran Tromer and Vinod Vaikuntanathan kindly agreed to serve on my PhD committee.

I am very grateful to Alessandro Chiesa for introducing me to zk-SNARKs, providing me with excellent advice and boundless support, but above all: for being a great friend. Ale's persistence, creativity, and high expectations, have been a profound influence on me that I am deeply thankful for.

I would like to thank Eli Ben-Sasson and Eran Tromer for being excellent mentors and friends. Eli and Eran hosted my research visits in Israel; my stays at Tel Aviv and Haifa were very productive, tons of fun, and I wish to return many times more. I also wish to thank Nickolai Zeldovich for the great research discussions and sage advice.

I am also thankful for my wonderful collaborators Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Daniel Genkin, Matthew Green, Shaul Kfir, Ian Miers, and Eran Tromer.

I wish to thank the members of MIT's Theory of Computation group, who made Stata's G5 and G6 light up with such energy and wonder: working with you has been amazing. I'm especially grateful to my officemates at G580 — it is you who made "Glorious Office" glorious. Finally, I would like to thank my family and my friends both here in the US and back at home.

Colophon

The most recent version of this thesis is available online: `https://madars.org/phd-thesis/`.
tru

Chapter 1

Introduction

Suppose that a remote computer has data that it is not willing to reveal (e.g. DNA database or cryptographic secrets). You have a program you would like to run on this data. The computer runs the program for you and gives you the output. How can the computer convince you (with a proof!) that the output is correct, without revealing the secret data?

This is a very general and important problem with many applications. Its essence is captured by fascinating notion of *zero-knowledge proofs* [GMR89], a powerful cryptographic tool with vast array of applications. Such applications include secure multi-party computation [GMW87a, BGW88], electronic voting [KMO01, Gro05, Lip11], anonymous credentials [BCKL08], group signatures [BW06, Gro06], among very many others. It would not be overemphasis to say that, within theoretical cryptography, zero-knowledge proofs are truly ubiquitous and indispensable.

Still, the impact of zero-knowledge proofs for building secure systems in practice has been modest at best. In particular, until very recently all reported uses of zero-knowledge proofs in the wild concerned proving very restricted NP statements: for example, knowledge of a discrete logarithm, subspace membership and similar facts of evidentially algebraic nature. Part of this can be explained by the economics of deploying new technology in the wild: often introducing a trusted third party in lieu of a proof system achieves users' security goals with lower anticipated cost.

In recent years this landscape has significantly changed: we have seen emergence of prototype implementations of generic zero-knowledge proof systems [PGHR13, BCG$^+$13a, BFR$^+$13], as well deployment of real-world systems, such as Zerocash [BCG$^+$14], that crucially rely on zero-knowledge proofs for complex NP statements.

One can point at two catalysts for this: first, a **new generation of zero-knowledge**

proofs where the proof length is much smaller than the complexity of the statement [Mic00, GW11, BCCT12, BCI+13]; and second, **new applications**, especially globally distributed ledgers like Bitcoin [Nak09], creating a demand for privacy of computation. The combination of two — concrete efficiency of modern proof systems, and lack of a universally trusted third party — means that, for certain applications, the economic calculus has ceased to be one-sided.

The goal of this thesis is to lower the cost of using zero-knowledge proofs in real-world systems. To elaborate on this goal and present the contributions of this thesis, the rest of this chapter is organized as follows. Section 1.1 gives basic background on *zero-knowledge Succinct Non-interactive ARguments of Knowledge* (zk-SNARK), the main object of study of this thesis. zk-SNARK constructions can be applied to a wide range of security applications, provided these constructions deliver good enough efficiency, and support rich enough functionality. This is particularly true in scenarios where trust is hard to come by, and in Section 1.2 we outline concrete applications of zk-SNARKs for globally distributed ledgers. Finally, the cost of deploying zk-SNARKs has two major components: the cost incurred by the proof system itself, and the price paid to instantiate the security model the proof system relies on. Working with my collaborators, I have contributed to lower both of these costs, and Section 1.3 outlines the main achievements of this thesis and how they contribute to the practical deployment of zero-knowledge proofs.

1.1 Introduction to zk-SNARKs

The main object of study of this thesis is a special kind of *Succinct Non-interactive ARgument of Knowledge* (SNARK). Concretely, most of the time we will be working with *publicly-verifiable preprocessing zero-knowledge* SNARK, or zk-SNARK for short. In this section we provide basic background on zk-SNARKs, provide an informal definition, compare zk-SNARKs with the more familiar notion of NIZKs. We now informally define zk-SNARKs for arithmetic circuit satisfiability, and refer the reader to, e.g., [BCI+13] for a formal definition.

For a field $\mathbb{F}$, an $\mathbb{F}$*-arithmetic circuit* takes inputs that are elements in $\mathbb{F}$, and its gates output elements in $\mathbb{F}$. We naturally associate a circuit with the function it computes. To model nondeterminism we consider circuits that have an *input* $\vec{x} \in \mathbb{F}^n$ and an auxiliary input $\vec{a} \in \mathbb{F}^h$, called a *witness*. The circuits we consider only have *bilinear gates*.[1] Arithmetic

[1] A gate with inputs $y_1, \ldots, y_m \in \mathbb{F}$ is *bilinear* if the output is $\langle \vec{a}, (1, y_1, \ldots, y_m) \rangle \cdot \langle \vec{b}, (1, y_1, \ldots, y_m) \rangle$ for

circuit satisfiability is defined analogously to the boolean case, as follows.

Definition 1.1.1. *The* arithmetic circuit satisfiability problem *of an* $\mathbb{F}$*-arithmetic circuit* $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ *is captured by the relation* $\mathcal{R}_C = \{(\vec{x},\vec{a}) \in \mathbb{F}^n \times \mathbb{F}^h \colon C(\vec{x},\vec{a}) = 0^l\}$*; its language is* $\mathscr{L}_C = \{\vec{x} \in \mathbb{F}^n : \exists \vec{a} \in \mathbb{F}^h \text{ s.t. } C(\vec{x},\vec{a}) = 0^l\}$*. We call* $\vec{x}$ *the input of* C *and* $\vec{a}$ *the witness of* C*.*

Given a field $\mathbb{F}$, a (publicly-verifiable preprocessing) **zk-SNARK** for $\mathbb{F}$-arithmetic circuit satisfiability is a triple of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{P}, \mathsf{V})$:

- $\mathsf{Gen}(1^\lambda, C) \to (\mathsf{pk}, \mathsf{vk})$. On input a security parameter λ (presented in unary) and an $\mathbb{F}$-arithmetic circuit C, the *key generator* Gen probabilistically samples a *proving key* pk and a *verification key* vk. Both keys are published as public parameters and can be used, any number of times, to prove/verify membership in $\mathscr{L}_C$.

- $\mathsf{P}(\mathsf{pk}, \vec{x}, \vec{a}) \to \pi$. On input a proving key pk and any $(\vec{x},\vec{a}) \in \mathcal{R}_C$, the *prover* P outputs a non-interactive proof π for the statement $\vec{x} \in \mathscr{L}_C$.

- $\mathsf{V}(\mathsf{vk}, \vec{x}, \pi) \to b$. On input a verification key vk, an input $\vec{x}$, and a proof π, the *verifier* V outputs $b = 1$ if he is convinced that $\vec{x} \in \mathscr{L}_C$.

A zk-SNARK satisfies the following properties.

Completeness. For every security parameter λ, any $\mathbb{F}$-arithmetic circuit C, and any $(\vec{x},\vec{a}) \in \mathcal{R}_C$, the honest prover can convince the verifier. Namely, $b = 1$ almost surely in the following experiment: $(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, C)$; $\pi \leftarrow \mathsf{P}(\mathsf{pk}, \vec{x}, \vec{a})$; $b \leftarrow \mathsf{V}(\mathsf{vk}, \vec{x}, \pi)$.

Succinctness. An honestly-generated proof π has $O_\lambda(1)$ bits and $\mathsf{V}(\mathsf{vk}, \vec{x}, \pi)$ runs in time $O_\lambda(|\vec{x}|)$. (Here, O_λ hides a fixed polynomial factor in λ.) Finally, succinctness is expressed as three efficiency requirements: (i) the generator and the prover must run in time that is polynomial in $|C|$; (ii) the verifier must run in time that is polynomial in $|x|$; and (iii) an honestly-generated proof must have size $\mathsf{poly}(\lambda)$.

Proof of knowledge (and soundness). If the verifier accepts a proof output by a bounded prover, then the prover "knows" a witness for the given instance. (In particular, soundness holds against bounded provers.) Namely, for every $\text{poly}(\lambda)$-size adversary $\mathcal{A}$, there is a $\text{poly}(\lambda)$-size extractor $\mathcal{E}$ such that $\mathsf{V}(\mathsf{vk}, \vec{x}, \pi) = 1$ and $(\vec{x},\vec{a}) \notin \mathcal{R}_C$ with probability $\mathsf{negl}(\lambda)$ in the following experiment: $(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, C)$; $(\vec{x}, \pi) \leftarrow \mathcal{A}(\mathsf{pk},\mathsf{vk})$; $\vec{a} \leftarrow \mathcal{E}(\mathsf{pk},\mathsf{vk})$.

some $\vec{a}, \vec{b} \in \mathbb{F}^{m+1}$. These include addition, multiplication, negation, and constant gates.

Perfect zero knowledge. An honestly-generated proof is perfect zero knowledge.Namely, there is a polynomial-time simulator S such that for all stateful distinguishers $\mathcal{D}$ the following two probabilities are equal:

$$\Pr\left[\begin{array}{c|r} (\vec{x},\vec{a}) \in \mathcal{R}_C & (\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, C) \\ \mathcal{D}(\pi) = 1 & (\vec{x},\vec{a}) \leftarrow \mathcal{D}(\mathsf{pk},\mathsf{vk}) \\ & \pi \leftarrow \mathsf{P}(\mathsf{pk},\vec{x},\vec{a}) \end{array}\right] \text{ and } \Pr\left[\begin{array}{c|r} (\vec{x},\vec{a}) \in \mathcal{R}_C & (\mathsf{pk},\mathsf{vk},\mathsf{trap}) \leftarrow \mathsf{S}(1^\lambda, C) \\ \mathcal{D}(\pi) = 1 & (\vec{x},\vec{a}) \leftarrow \mathcal{D}(\mathsf{pk},\mathsf{vk}) \\ & \pi \leftarrow \mathsf{S}(\mathsf{trap},\vec{x}) \end{array}\right] .$$

(the probability that $\mathcal{D}(\pi) = 1$ on an honest proof) (the probability that $\mathcal{D}(\pi) = 1$ on a simulated proof)

Both proof of knowledge and zero knowledge are essential to many interesting uses of zk-SNARKs. Indeed, if we consider circuits C that verify assertions about cryptographic primitives (such as using a knowledge of SHA-256 pre-image as a binding commitment). Thus it *does not suffice* to merely know that, for a given input $\vec{x}$, a witness for $\vec{x} \in \mathscr{L}_C$ *exists*. Instead, proof of knowledge ensures that a witness can be efficiently found (by extracting it from the prover) whenever the verifier accepts a proof. As for zero knowledge, it ensures that a proof leaks no information about the witness, beyond the fact that $\vec{x} \in \mathscr{L}_C$.

zk-SNARKs are related to a familiar cryptographic primitive: *non-interactive zero-knowledge proofs of knowledge* (NIZKs). Both zk-SNARKs and NIZKs require a one-time trusted setup of public parameters (proving and verification keys for zk-SNARKs, and a common reference string for NIZKs). Both provide the same guarantees of completeness, proof of knowledge, and zero knowledge. The difference lies in efficiency guarantees. In a NIZK, the proof length and verification time depend on the NP language being proved. For instance, for the language of circuit satisfiability, the proof length and verification time in [GOS06b, GOS06a] are linear in the circuit size. Conversely, in a zk-SNARK, proof length depends only on the security parameter, and verification time depends only on the instance size (and security parameter) but not on the circuit or witness size.

Thus, zk-SNARKs can be thought of as "succinct NIZKs", having short proofs and fast verification times. Succinctness comes with a caveat: known zk-SNARK constructions rely on stronger assumptions than NIZKs do. Yet, succinctness is a very attractive property for a practical viewpoint, which may enable applications for which NIZKs would be too inefficient. In fact, as we explain in Section 1.2, recent developments in the realm of distributed systems have created applications for which both non-interactivity and succinctness are crucial requirements.

1.2 zk-SNARKs for distributed ledgers

Bitcoin is the first digital currency to achieve widespread adoption. The currency owes its rise in part to the fact that, unlike traditional e-cash schemes [Cha82, CHL05, ST99], it requires no trusted parties. Instead of appointing a central bank, Bitcoin uses a distributed ledger known as the *blockchain* to store transactions carried out between users. Because the blockchain is massively replicated by mutually-distrustful peers, the information it contains is public.

The characteristics of distributed ledgers, like Bitcoin, mean that the following proof system properties are especially useful:

- **Non-interactivity.** Because Bitcoin is globally distributed, the parties of the system might simply not be online at the same time. This is especially true, if the statement to be proved has to be verified by the participant in the system. For example, this is the case when validity of transaction depends on secret information held by the prover; for the same reason digital signatures are more appropriate than identification protocols.

- **Succinctness.** The *raison d'être* for blockchain is to enable global consensus between untrusting parties, and thus the state of the blockchain is duplicated by every participant node. The associated storage, network communication and computational complexity requirements raise significant scalability challenges; in fact, Bitcoin core developers argue that, for the health of the network, the size of individual Bitcoin blocks should not exceed a couple megabytes, and individual transactions: couple hundreds of bytes. Any additional data we might wish to store in the blockchain, including cryptographic proofs, must be commensurate with this. Moreover, to ensure fast synchronization times and reduce the risk of "orphan" blocks, the transactions should be cheap to verify (say, order of milliseconds).

We now argue, by a way of example, that a cryptographic proof system satisfying these two properties can be very useful for improving the scalability of distributed ledgers.

Reducing transaction sizes. Even without considering zero-knowledge, zk-SNARKs are an effective tool for improving scalability of systems due to their succinctness. For example, Bitcoin Pay to Script Hash (P2SH) transactions require that two pieces of information are permanently stored on the blockchain: script matching the script hash and data which makes the script evaluate to true. Consequently, more complex (and more interesting)

P2SH transactions take more of the valuable block space, require proportionally fees and proportionally lower throughput.

However, P2SH spends can be scaled up by zk-SNARK proofs: we replace the script and its data by a proof that script/data *exists* making the transaction valid; a condition the algorithm A could check. No matter what the script size or the data size, the Blockchain would only need to contain the script hash and 300 byte proof.

While best exemplified by P2SH spends, this idea can be generalized to handle other transaction types as well. For example, to reduce size of Pay to Pubkey Hash, one would *omit* all ECDSA signatures, listing the UTXO inputs (UTXOs are transactions that have unspent outputs, and thus cannot be pruned from the blockchain) and a SNARK proof attesting an existence of signature for each of them.

Improving the security of SPV clients. Clients following the Simple Payment Verification (SPV) protocol lack the ability to authenticate whether or not their view of the blockchain, provided by an untrusted source, complies with the consensus rules. Two major directions in this area are proofs of authorized UTXO set modifications (for proving absence of fraud), and fraud proofs (for establishing presence of it). Both would benefit from efficient zk-SNARKs: in the former case, each block would be augmented by a compact SNARK proof certifying correctness of the block; in the latter case a zk-SNARK would ascertain *existence* of blockchain invariant violation, without burdening the client with its full verification. We remark that many pruned UTXO set and checkpointing proposals rely on similar techniques, and refer the reader to Bryan Bishop's excellent overview talk from Scaling Bitcoin, Montreal [Bis15] for more information.

Preventing centralization. A proposal to prevent mining centralization, non-outsourceable scratch-off puzzles [MSKK15], rely on zk-SNARKs as the main ingredient in their technical toolbox. Very roughly speaking, the proof-of-work system is modified so that a winning solution s (e.g. a nonce that produces hash value under the difficulty threshold), can be transformed into a different solution s', such that: (*a*) s' cannot be linked back to the original solution s; and (*b*) s' can be spent in a different coinbase transaction. This way, mining pools have no incentive to form: any rational miner within the pool, would just spend a solution himself, rather than sharing it with the pool operator. The zero-knowledge property of zk-SNARKs ensures that such "traitors" cannot be caught.

Improving block propagation. An alternative proposal uses succinct verification to speed-up block propagation time, as follows. In parallel with mining on the block header for a new block, the miner would also produce a zk-SNARK proof for the following statement: "each transaction belonging to `hashMerkleRoot` is valid with respect to `hashPrevBlock`". Doing so is a competitive advantage for the miner: easy to verify blocks enjoy fast propagation and decreased orphaning rates.

Preventing economic obstacles. Threat to currency's fungibility is a fundamental threat to its adoption and, thus, scalability. A definite approach to ensure fungibility is making all transactions private, while keeping the economic invariants intact. Here the blockchain would store zero-knowledge proofs that unspent coins were properly transferred, but would reveal nothing about the parties or amounts. This is the approach taken by Zerocash [BCG+14].

1.3 Contributions of this thesis

The cost of deploying zk-SNARKs has two major components: the cost incurred by the proof system itself, and the price paid to instantiate the security model the proof system relies on. This thesis contributes to reducing both of these costs:

Cost of the security model. For many practical scenarios it is crucial that proofs be non-interactive and succinct. In the standard model, non-interactive zero-knowledge (NIZK) proofs do not exist for languages outside BPP (even with just computational soundness). However, if the security model includes a trusted party, available for a one-time setup phase, then NIZKs exist for all languages in NP [BFM88, GO94]. Soundness of the NIZK depends on this trusted setup: if public parameters are not correctly generated, or if the trusted party's secret internal randomness is revealed, an attacker could convince the verifier of false NP statements without being detected. In Chapter 3 we show how public parameters for a class of NIZKs can be generated by a concretely-efficient multi-party protocol, such that if at least one of the parties is honest, then the result is secure and can be subsequently used for generating and verifying numerous proofs without any further trust.

Cost of the proof system. We have designed and built an open-source cryptographic library, called **libsnark**, that provides efficient implementations of state-of-the-art zero-knowledge proof constructions. Our library is the fastest and most comprehensive suite of zero-knowledge proofs currently available. The **libsnark** library was organically built and extended to achieve the results reported in half a dozen papers [BCG+13a, BCTV14a, BCTV14c, CTV15, BCG+14, BCG+15], each of which report about pieces of the ever-growing library. To this date, **libsnark** remains the fastest and most comprehensive suite of zero knowledge proofs, and the implementation itself entails many algorithmic and engineering details. In Chapter 2 we provide unified view of this cryptographic library, focusing on the high-performance engineering aspects.

Working in tandem, these two contributions have achieved industrial impact, and are the main efficiency enablers for Zerocash, a privacy-preserving payment system.

Chapter 2

libsnark: a software library for succinct zero-knowledge proofs

The goal of the **libsnark** library is to be *the* cryptographic library for succinct zero-knowledge proofs. We built the **libsnark** library to be appealing to two types of users: non-cryptographers who want high-level interfaces to enforce security policies; and researchers who want to build new cryptographic primitives based on SNARKs. We can cite more than fifty academic papers that build upon **libsnark**, and more than a couple industrial start-ups that use **libsnark**, as indirect evidence for making a solid showing towards this goal.

Development principles. When building **libsnark** we made careful choices about structure of the library, and aimed to ensure maximum generality of all its components, including, algebraic routines, NP relations, NP reductions, and proof-system backends and frontends. Using software engineering terms, we aimed for loose coupling (each component of the library should have the least possible dependencies), and high cohesion (single responsibility principle: do one thing and one thing well). In hindsight, the abstractions achieved in **libsnark** have helped us ensure fast iteration, for example, owing to careful reuse of general components, we were able to implement the [Gro16] proof system on the same day the preprint appeared on ePrint.

The libsnark stack. At a high level, the **libsnark** library can be decomposed as pictured in Figure 2.1. The foundation of the library is its shared algorithmic core; this includes implementations of algebraic objects, in particular, finite fields and bilinear groups, and algorithmic routines for performing essential algebraic transforms (e.g. polynomial evaluation and interpolation, or multi-scalar multiplication). The shared algorithmic core

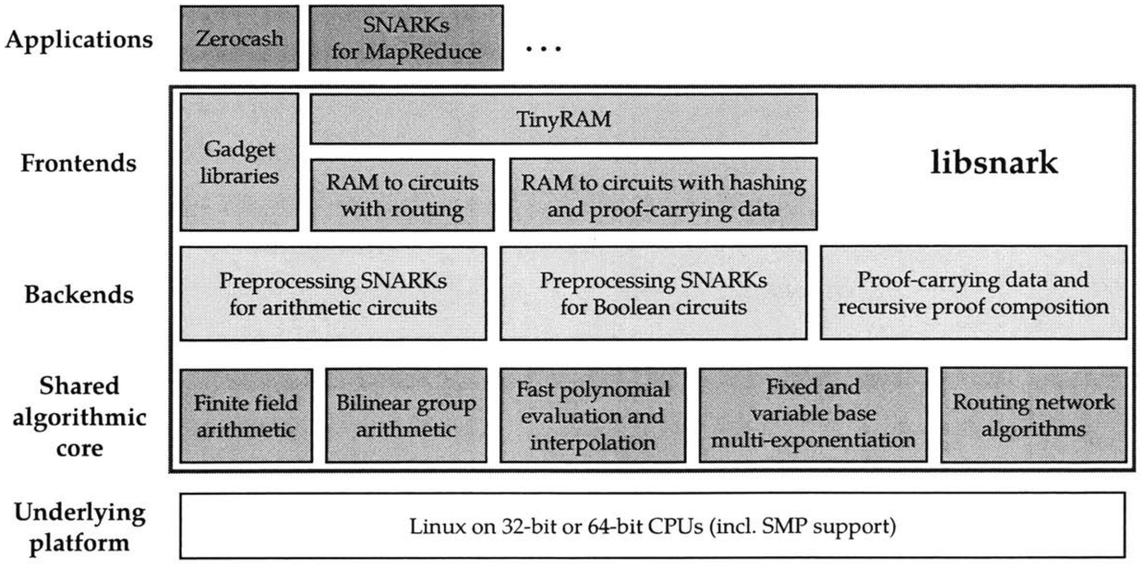

Applications
Zerocash
SNARKs for MapReduce
...
Frontends
Gadget libraries
TinyRAM
RAM to circuits with routing
RAM to circuits with hashing and proof-carrying data
libsnark
Backends
Preprocessing SNARKs for arithmetic circuits
Preprocessing SNARKs for Boolean circuits
Proof-carrying data and recursive proof composition
Shared algorithmic core
Finite field arithmetic
Bilinear group arithmetic
Fast polynomial evaluation and interpolation
Fixed and variable base multi-exponentiation
Routing network algorithms
Underlying platform
Linux on 32-bit or 64-bit CPUs (incl. SMP support)

Figure 2.1: Overview of the **libsnark** stack. The three major pieces of the library are proof system frontends, proof system backends, and the shared algorithmic core.

also implements routing networks. Proof-system backends build upon the algorithmic core, and achieve zk-SNARKs for highly rigid relations, such as, circuit satisfiability. Finally, proof-system frontends implement and expose zk-SNARKs for higher-level relations, such as, satisfiability of RAM programs. In particular, **libsnark** targets a custom CPU architecture called TinyRAM that is specifically built to be easily verified by zk-SNARKs. Internally, the frontends use "gadget" libraries to build circuits for checking these higher level languages, and then apply backend proof systems for circuit satisfiability.

As mentioned above, the components of the **libsnark** stack are independent in that each can be useful without the others:

- If one designed a pre-processing SNARK for circuits that is more efficient than ours, it could be plugged into **libsnark**, and benefit from frontend proof systems (e.g. our circuit generator for TinyRAM programs). Historically, this was the case for [PGHR13] and [Gro16] proof-systems, which have since been incorporated in **libsnark**.

- If one designed a different CPU architecture than TinyRAM, it would only require specifying the CPU transition function to benefit from our generic frontend reductions. In particular, RAM to circuits reduction with routing, and RAM to circuits reduction with hashing and proof-carrying data work for any load-store architecture.

- If one had an NP problem already represented via arithmetic circuit satisfiability (for instance, this is simple to achieve when considering "structured" computational problems such as matrix multiplication or evaluating FFTs) then there is no need to use a frontend proof system, so one could directly invoke our zk-SNARK.
- If one discovered a more efficient pairing-friendly group, its implementation could be used with any of our backend proof systems. This has been both the case historically, and is also true for industrial users of **libsnark**.
- If one discovered a more efficient multi-exponentiation algorithm, or more efficient way to solve polynomial evaluation and interpolation problems, these improvements would benefit all upper layers.

Of course, too much generality can harm performance and make systems hard to maintain. Fortunately, the **libsnark** decomposition has worked well in practice — to the best of our knowledge the combined reductions are quite tight, and breaking the layers of abstraction, or fusing individual components, would give insignificant efficiency gains.

In addition to the three component groups (proof system frontends, backends, and algorithmic core), **libsnark** also includes other auxiliary routines and data structures. In particular, the library has components for serialization, profiling, random number generation, and implements Merkle trees, sparse vectors, knowledge commitments, among others. In this Chapter we focus of the algorithmic aspects of **libsnark**, and thus will not discuss these routines or data structures in further detail.

Assumptions. To ensure soundness and zero-knowledge, **libsnark** relies on cryptographic assumptions (mostly, knowledge-of-exponent assumptions in bilinear groups, and standard assumptions about cryptographic hash functions), assumptions about randomness, and the execution environment. The library intentionally does not target resistance against side-channel attacks. In particular, as all proof systems in **libsnark** are publicily verifiable, these attacks do not matter for the SNARK verifiers; in the backend proof systems we reasonably expect that the frontend-to-backend reduction will not be side-channel free. Therefore, while nice to have, a side-channel resistant backend alone would not be effective for building side-channel resistant *systems*.

Code base. The **libsnark** library comprises about 26 000 lines of templatized C++, about 350 lines of hand-written x86-64 assembly, and a little over than 500 lines of Python. The Python code is mostly used to emit test cases for unit testing, and assembly routines, used for high-performance implementations of finite field arithmetic and binary heaps, have portable C++ alternatives.

Our cryptographic library primarily targets Linux (owing to the general availability of high-performance Linux clusters), but **libsnark** has few external dependencies, in principle, it should be easy to port **libsnark** to other systems. In fact, to achieve a demonstration at CRYPTO 2013 [BCG+13a], an earlier version of **libsnark** was ported to Android phones with ARM CPUs. Some downstream users of **libsnark** have ported parts of the library to Windows, and macOS.

Roadmap. The rest of this chapter is organized as follows. In Section 2.1 we give a 10 000 ft glimpse of a typical pre-processing SNARK backend; this view puts key algebraic problems in context, and provides the basis for discussing **libsnark**'s shared algorithmic core. In Section 2.2 we survey the zk-SNARK backends available in **libsnark**, and in Section 2.3 we survey the zk-SNARK frontends in libsnark, focusing our attention to `gadgetlib1`, the "gadget" library used to implement all our frontend reductions. We devote Section 2.4 to discuss the library's shared algorithmic core. Finally, in section Section 2.5 we discuss the impact of **libsnark**, and consider some future optimizations.

2.1 A taste of pre-processing zk-SNARKs

The language natively supported by many modern pre-processing zk-SNARK constructions is that of *quadratic arithmetic programs* (QAPs), introduced by Gennaro et al. [GGPR13].

Definition 2.1.1. *A* ***quadratic arithmetic program (QAP)*** *of size m and degree d over a field $\mathbb{F}$ is a quadruple $(\vec{A}, \vec{B}, \vec{C}, Z)$, where each of $\vec{A}$, $\vec{B}$ and $\vec{C}$ is a vector of $m+1$ polynomials in $\mathbb{F}^{\leq d-1}[z]$, and $Z(z) \in \mathbb{F}[z]$ is a monic polynomial of degree exactly d.*

Definition 2.1.2. *The* ***satisfaction problem*** *of a size-m QAP $(\vec{A}, \vec{B}, \vec{C}, Z)$ is the relation $\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$ of pairs $(\vec{x}, \vec{a}) \in \mathbb{F}^n \times \mathbb{F}^m$ satisfying the following conditions: (a) $n \leq m$ and $x_i = a_i$ for $1 \leq i \leq n$ (that is, $\vec{a}$ extends $\vec{x}$); and (b) $Z(z)$ divides the polynomial $(A_0(z) + \sum_{i=1}^m a_i A_i(z)) \cdot (B_0(z) + \sum_{i=1}^m a_i B_i(z)) - (C_0(z) + \sum_{i=1}^m a_i C_i(z))$.*

Gennaro et al. [GGPR13] showed that QAP satisfiability problem is NP-complete, in particular, they showed that arithmetic circuit satisfiability can be efficiently reduced to QAP satisfiability. We now give a very brief sketch for a proof system targeting the QAP satisfiability problem. This proof system first appeared in Ben-Sasson et al. [BCG+13a] and follows the "linear PCP" approach put forth by Bitansky et al. [BCI+13].

We assume existence of a *linear-only* cryptographic encoding of $\mathbb{F}_r$, that is, an encoding $\mathsf{Enc}\colon \mathbb{F}_r \to \{0,1\}^*$ with the following properties:

- it is easy to verify that an alleged encoding is indeed in the image of Enc;
- given encodings $e_1, \ldots, e_n$, it is feasible to check whether quadratic relationship holds between the preimages of the encodings (that is, whether a degree-2 polynomial $P(x_1, \ldots, x_n)$ evaluates to 0 on the vector of preimages $x_1, \ldots, x_n$);
- Enc admits efficient computation of all $\mathbb{F}_r$-linear homomorphisms, while making sure that other operations are computationally intractable ("up to" the information leaked by the quadratic predicates); and
- it provides a certain notion of one-way security to encoded elements.

We defer to [BCI$^+$13] for full technical definition of linear-only encodings.

Construction 2.1.3 (A SNARK for QAPs (sketch)). *The SNARK generator, prover and verifier work as follows:*

- ***Generator.*** *On input QAP $(\vec{A}, \vec{B}, \vec{C}, Z)$, the SNARK generator G first picks a random field element $\tau \in \mathbb{F}_r$. The generator uses τ to compute and output linear-only encodings* $\mathsf{Enc}(A_i(\tau))$, $\mathsf{Enc}(B_i(\tau))$, $\mathsf{Enc}(C_i(\tau))$, $\mathsf{Enc}(Z(\tau))$ *($1 \leq i \leq m$), as well as d linear-only encodings* $\mathsf{Enc}(1)$, $\mathsf{Enc}(\tau)$, $\ldots$, $\mathsf{Enc}(\tau^{d-1})$. *The generator also picks three random field elements $\alpha, \beta, \gamma \in \mathbb{F}_r$ and outputs linear-only encodings of a random linear combination of $A_i(\tau)$, $B_i(\tau)$, $C_i(\tau)$; more precisely, G also outputs the linear-only encodings* $\mathsf{Enc}(\alpha A_i(\tau) + \beta B_i(\tau) + \gamma C_i(\tau))$ *for $1 \leq i \leq m$, as well as* $\mathsf{Enc}(\alpha)$, $\mathsf{Enc}(\beta)$, $\mathsf{Enc}(\gamma)$.
- ***Prover.*** *On input instance $\vec{x}$ and satisfying witness $\vec{a}$, the SNARK prover P uses $\vec{a}$ to compute the vector of coefficients $\vec{h}$ of the polynomial*

$$H(z) := \frac{(A_0(z) + \sum_{i=1}^{m} a_i A_i(z)) \cdot (B_0(z) + \sum_{i=1}^{m} a_i B_i(z)) - (C_0(z) + \sum_{i=1}^{m} a_i C_i(z))}{Z(z)} \,.$$

As $(\vec{x}, \vec{a}) \in \mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, the division above has no remainder. After having computed $H(z)$, the prover uses the linear-only encodings output by the SNARK generator G to obtain the following *five components of the proof π:*

$$\pi_{\mathsf{A}} := \mathsf{Enc}\left(\sum_{i=n}^{m} a_i A_i(\tau)\right) \qquad \pi_{\mathsf{B}} := \mathsf{Enc}\left(\sum_{i=n}^{m} a_i B_i(\tau)\right)$$

$$\pi_{\mathsf{C}} := \mathsf{Enc}\left(\sum_{i=n}^{m} a_i C_i(\tau)\right) \qquad \pi_{\mathsf{H}} := \mathsf{Enc}\left(\sum_{i=0}^{d-1} h_i \tau^i\right) = \mathsf{Enc}\left(H(\tau)\right)$$

$$\pi_{\mathsf{K}} := \mathsf{Enc}\left(\sum_{i=n}^{m} a_i(\alpha A_i(\tau) + \beta B_i(\tau) + \gamma C_i(\tau))\right)$$

- ***Verifier.*** *On input instance $\vec{x}$ and purported proof π, the zk-SNARK verifier V does the following. It first checks that each component of the proof, that is, π_{A}, π_{B}, π_{C}, π_{H} and π_{K} are all in the image of* Enc. *Next, it performs two quadratic checks: first, it uses the encodings of α, β, and γ, output by the generator G to test whether* $\mathsf{Enc}(\alpha) \cdot \pi_{\mathsf{A}} + \mathsf{Enc}(\beta) \cdot \pi_{\mathsf{B}} + \mathsf{Enc}(\gamma) \cdot \pi_{\mathsf{C}} = \pi_{\mathsf{K}}$ *holds. Next, it uses the instance $\vec{x}$ to compute three encodings* $\chi_{\mathsf{A}} = \mathsf{Enc}(A_0(\tau) + \sum_{i=1}^{m} x_i A_i(\tau))$, $\chi_{\mathsf{B}} = \mathsf{Enc}(B_0(\tau) + \sum_{i=1}^{m} x_i B_i(\tau))$ *and* $\chi_{\mathsf{C}} = \mathsf{Enc}(C_0(\tau) + \sum_{i=1}^{m} x_i C_i(\tau))$, *and tests whether* $(\chi_{\mathsf{A}} + \pi_{\mathsf{A}}) \cdot (\chi_{\mathsf{B}} + \pi_{\mathsf{B}}) = \pi_{\mathsf{H}} \cdot \mathsf{Enc}(Z(\tau)) + (\chi_{\mathsf{C}} + \pi_{\mathsf{C}})$. *The verifier accepts if and only if all these tests succeed.* □

We now give a highly compressed overview of why the proof system above is complete, sound, and explain that with minor modifications it can also be made zero-knowledge. The main purpose of this sketch is to motivate the cryptographic and algorithmic problems encountered by G, P and V, and we refer the reader to [BCG^{+}13a],[BCI^{+}13] for details.

Completeness. We can convince ourselves that the proofs output by an honest prover always satisfy the verifier's checks. Indeed, their elements are in the domain of Enc (so preimage checks are satisfied), the element π_{K} is indeed a linear combination of π_{A}, π_{B}, and π_{C}, with coefficients α, β, γ (so verifier's first quadratic test is satisfied), and the verifier's second quadratic check is exactly implied by the satisfaction of QAP. More precisely, $(A_0(z) + \sum_{i=1}^{m} a_i A_i(z)) \cdot (B_0(z) + \sum_{i=1}^{m} a_i B_i(z)) - (C_0(z) + \sum_{i=1}^{m} a_i C_i(z)) = H(z) \cdot Z(z)$ is a polynomial identity, so it continues to hold when the formal variable z is replaced by the generator's concrete choice of $z := \tau$.

Knowledge soundness. As part of its one-way security, we require that linear-only encoding Enc satisfies a certain notion of extractability: for any algorithm $\mathcal{A}$ that, on input encodings $\mathsf{Enc}(u_1), \ldots, \mathsf{Enc}(u_n)$, outputs an encoding e in the image of Enc, there exists an extractor $\mathcal{E}_{\mathcal{A}}$, which on the same inputs as $\mathcal{A}$ (including $\mathcal{A}$'s randomness tape), outputs a linear combination $c_1, \ldots, c_n$, "explaining" the linear-only homomorphism applied by $\mathcal{A}$. That is, $\mathcal{E}_{\mathcal{A}}$ outputs $\vec{c}$ for which the following linear relationship holds: $e = c_1 \cdot \mathsf{Enc}(u_1) + \cdots + c_n \cdot \mathsf{Enc}(u_n)) = \mathsf{Enc}(c_1 \cdot u_1 + \cdots + c_n \cdot u_n)$.

The output of the zk-SNARK prover algorithm P consists of six valid encodings, therefore, under the aforementioned security assumption, there exists an extractor $\mathcal{E}_P$, outputting linear combinations that "explain" each of the proof elements. In particular, elements π_A, π_B, π_C, and π_K allows one to recover the witness vector $\vec{a}$, whereas the element π_H, allows one to recover the coefficient vector $\vec{h}$. Because verifier's checks pass, the vectors $\vec{a}$ and $\vec{h}$ jointly satisfy the QAP polynomial divisibility equation for $z = \tau$. With more work, and relying on the hiding properties of Enc, one can show that $\vec{a}$ and $\vec{h}$ make the equation to be a polynomial identity, and therefore $(\vec{x}, \vec{a}) \in \mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$.

Zero-knowledge. To make our toy proof system zero-knowledge one can apply the following trick: add three random multiples of $Z(\tau)$ to π_A, π_B, and π_C, respectively, and adjust the computations of π_K and π_H to be consistent with these choices. This choice makes π_A, π_B, and π_C uniformly random, while the proof elements π_K and π_H are uniquely determined by the values of π_A, π_B, and π_C, and the satisfiability of verifier's quadratic check. One can also prove that such strategy does not spoil the knowledge soundness of our proof system; this comes from the fact that $Z(z)$ has higher degree than all polynomials $A_i(z)$, $B_i(z)$, and $C_i(z)$, and therefore the extractor can uniquely cancel out the contribution of the randomization terms.

The above sketch can be made formal, and we do so in Appendix A. Apart from making sure that the proof system can be made work in abstract, there are three natural questions that need to be answered to make this proof system work in practice:

1. **Security.** Do there exist linear-only encodings Enc for which the necessary security assumptions are plausible?

2. **Efficiency.** How can one efficiently instantiate the algorithms G, P, and V?

3. **Expressivity.** Does the language of QAP satisfiability capture useful problems? That is, given an "interesting" NP relation, can we write down the polynomials $(\vec{A}, \vec{B}, \vec{C}, Z)$?

The answers to the first two questions are both "yes", and we deal with those in Section 2.4. Jumping ahead, the map $\mathsf{Enc} : x \to (g^x, g^{\alpha x})$ for certain bilinear groups over elliptic curves is believed to satisfy the necessary security and functionality requirements; this motivates **libsnark**'s cryptographic core. Similarly, the main computations performed by the zk-SNARK generator and prover are special instances of widely studied questions about polynomial evaluation interpolation, and multi-scalar multiplication. These motivate the bulk of **libsnark**'s algorithmic core.

In the next two Sections we focus our attention on the question about expressivity: in Section 2.2 we will see that span programs capture all of NP via elegant reductions from circuit satisfiability problems, and in we will show that there are modular ways to "program" circuits, yielding to efficient and generic implementations.

2.2 Backend relations

As mentioned in Section 2.1, most modern pre-processing zk-SNARK backends are built for various kinds of polynomial span programs, such as, quadratic arithmetic programs [GGPR13] or square span programs [DFGK14]. As noticed by Gennaro et al., questions about a witness vector w simultaneously satisfying n polynomial constraints can often be reduced to a question about divisibility of degree n polynomials. Relying on Schwartz–Zippel lemma, the divisibility question can be probabilistically verified by performing the check at just one randomly chosen point.

For example, suppose you have n constraints of the form $\langle a_{i,j}, w_j \rangle \cdot \langle b_{i,j}, w_j \rangle = \langle c_{i,j}, w_j \rangle$, specified by three constant matrices A, B and C with entries $a_{i,j}$, $b_{i,j}$, $c_{i,j}$. Such constraint systems naturally capture arithmetic circuits with bilinear gates: when w is a complete assignment to the circuit's m wires, the inner products $\langle a_{i,j}, w_j \rangle$ and $\langle b_{i,j}, w_j \rangle$ bundle the left and right inputs of the j-th bilinear gate, and $\langle c_{i,j}, w_j \rangle$ can be chosen to equal the j-th gates output wire.

Now pick n field elements $\sigma_1, \dots, \sigma_n$, and let $A_i(z)$, $B_i(z)$ and $C_i(z)$ be the unique polynomials of degree less than n satisfying $A_i(\sigma_j) = a_{i,j}$, $B_i(\sigma_j) = b_{i,j}$ and $C_i(\sigma_j) = c_{i,j}$ for all $1 \leq j \leq n$. One can see that n original constraints are satisfied if and only if the polynomial $\mathsf{QAP}(z) := (\sum_{i=1}^{m} w_i A_i(z)) \cdot (\sum_{i=1}^{m} w_i B_i(z)) - (\sum_{i=1}^{m} w_i C_i(z))$ vanishes on the set $\{\sigma_1, \dots, \sigma_n\}$, or, equivalently, $\mathsf{QAP}(z)$ is divisible by $Z(z) := \prod_{i=1}^{n}(z - \sigma_i)$.

The above reduction, due to Gennaro et al. [GGPR13], is particularly tight: a circuit with n gates gives rise to a single equation about polynomials of degree at most n. Accordingly, the main algebraic tools of **libsnark** backend proof systems involve solving polynomial evaluation and interpolation problems, whereas the main cryptographic tools facilitate evaluation and divisibility checking of encoded polynomials.

The approaches to instantiating cryptographic encodings and performing the divisibility checks, vary a great deal both in terms of the performance and properties achieved, as well as in terms of the cryptographic assumptions. For example, the `r1cs_ppzksnark` proof system, based on [PGHR13, BCTV14c], relies on q-type knowledge-of-exponent assumptions and requires 5 pairing-based checks: 3 check validity of cryptographic encodings, 1 check that answer lies in correct polynomial span, and 1 check the polynomial divisibility. In contrast, the [Gro16] relies on much strong generic group model and performs just one pairing-based check. We refer reader to **libsnark**'s performance comparison[1] for further details.

2.3 Proof system frontends

The goal of frontend proof systems is to enable obtaining zk-SNARKs for high level, developer-friendly NP languages. As all **libsnark** backends provide zk-SNARKs for (essentially) circuit satisfiability, the proof system frontends work by directly translating these higher-level specifications into arithmetic circuits, or by providing universal circuits whose inputs are high-level representations. In **libsnark** example of a direct translation is `gadgetlib1`, the C++ library of composable circuit "gadgets", whereas the latter approach is realized by our universal circuits for load-store RAM architectures.

The RAM reductions are covered in extensive detail in [BCG$^+$13a, BCTV14c, BCTV14a], while `gadgetlib1`, the main software engineering enabler for these reductions, has received more narrow exposition. Therefore, we devote most of this section to describing `gadgetlib1` and its main design principles.

2.3.1 `gadgetlib1`: library for "programming" circuits

The performance of backend zk-SNARK generator and prover algorithms highly depend on the complexity of the particular NP relation $\mathcal{R}$. More precisely, both the generator and prover work in time and space that is quasilinear in size of the circuit deciding $\mathcal{R}$. Therefore, it is highly desirable to obtain circuits of smallest possible size: indeed, any inefficiencies in the circuit representation will be paid for by every single prover.

The compiler approach. Prior work has explored a particular route to obtaining circuits: write a program deciding $\mathcal{R}$ in some high-level programming language, such as, C, and use a circuit generator to obtain an arithmetic circuit for the backend zk-SNARK. The main

[1] https://github.com/scipr-lab/libsnark/blob/master/libsnark/zk_proof_systems/ppzksnark/README.md

benefit of this approach is developer convenience: developers can use a familiar language, and benefit from existing tooling, testing infrastructure, etc.

Importance of non-determinism. However, as we show in Section 2.3.3 this approach can yield results that are an integer multiple away from tailored approaches. At high level, the main reason for this is inefficient use of non-determinism: for many functions f, computing the value of f is significantly more complicated, than checking whether an already computed answer is correct. In particular, checkers can benefit from non-deterministic auxiliary inputs, while these inputs are not available when reducing from a high-level program which *computes* the answer.

To illustrate the benefits of non-determinism consider the task of deciding whether or not two n-element lists are sorted copies of each other. The most efficient circuits for sorting are of size $O(n \log^2 n)$; to our knowledge this is the best approach that does use non-determinism.[2] However, if our relation can depend on non-deterministic inputs, a much better $O(n \log n)$ approach is possible: use a routing network [Ben65, Wak68] to implement a permutation between the two lists. Here the non-deterministic inputs are switch settings for the routing network; these need to be computed separately by the zk-SNARK prover, but cost of doing so is negligible, compared to $O(\log n)$ multiplicative blow-up in the cryptographic cost for the sorting network.

The design of `gadgetlib1`. To obtain maximally efficient low-level representations of interesting NP relations, we built a library for wiring non-deterministic circuit "gadgets", emphasizing modularity and non-determinism. The `gadgetlib1` library is a suite of optimized subcircuit "gadgets" together with means of effectively composing, testing, and debugging them. Each gadget is a pair of two algorithms: *instance map*, which generates the circuit wiring, and *witness map*, which computes the wire values of the gadget, including assignments to non-deterministically set wires. It is exactly the use of non-determinism that sets apart gadget from a traditional arithmetic circuit, where all wire values can be computed directly from the circuit wiring.

Importance of modularity. An obvious reason for building circuits out of modular parts is, of course, programmer convenience and work saved by component reuse. However, the use of non-determinism presents additional challenges to ensure soundness. For deterministic circuits completeness and soundness are naturally linked: by computing the unique answer each gate contributes to enforcing correctness. Non-deterministic circuits

[2] There exist asymptotically smaller sorting networks, e.g., $O(n \log n)$-sized AKS sorting network [AKS83] or the recent zig-zag sort [Goo14]. However, the O notation for all of these constructions hide large constant terms that dwarf $\log n$ for all practically relevant n.

stand in stark contrast with this: an existence of a correct wire assignment does not imply a non-existence of incorrect one. Let us highlight this by an example.

Consider the task of verifying integer division: given three inputs a, b and c accept if and only if $c = \lfloor a/b \rfloor$. Let us further assume that a and b are restricted to some range, e.g. $0 \leq a, b < 2^k$. Standard deterministic circuits computing the result of the division are of size $O(k^2)$. A non-deterministic solution would be to guess remainder r, and check that the following two conditions are satisfied: $a = bc + r$ and $0 \leq r < 2^k$; both of these conditions together can be checked by using an $O(k)$-gate circuit. In addition to relying on "native" field arithmetic, provided by arithmetic circuits (assuming that field characteristic is higher than 2^r, this lets one compute the value bc using just one bilinear gate), we use further non-determinism to ensure that $0 \leq r < 2^k$. That is, we non-deterministically guess the k bits b_i of r, and enforce that $r = \sum_{i=0}^{k-1} b_i 2^i$, as well as $b_i \in \{0, 1\}$. The latter can be done by one bilinear constraint: $b_i(1 - b_i) = 0$.

In the deterministic case, if any gate is forgotten, the circuit will likely yield an incorrect answer. In the non-deterministic case, the witness map will correctly compute $c := \lfloor a/b \rfloor$, even if one of the checks is absent. Of course, if any of the two checks are absent, a careful choice of r can satisfy the gadget for *any* malicious answer c. To avoid creating gadgets with vacuous guarantees `gadgetlib1` emphasizes modularity to make security audits easier, and use of negative tests (checking that unsatisfiable assignments are indeed rejected by the non-deterministic circuit). Ideally, the counter-example checking would be done by static analysis techniques (e.g. SMT solvers), and soundness would be assured by formal verification methods; we leave these improvements for further work.

2.3.2 RAM reductions and TinyRAM

As we will see in Section 2.3.3, the direct "gadget" approach described in the previous section lets one obtain circuits of very small size. However, this efficiency comes with two drawbacks. First, directly wiring "gadgets" requires high programmer effort — our use-case of `SHA-256` took a week of effort from two PhD students with high expertise in practical proof systems, whereas specifying this relation in a high-level programming language would dramatically decrease the required effort and expertise.

Second, direct approaches necessarily require knowledge of the particular relation ahead of time, as the zk-SNARK setup algorithm immutably encodes the relation in the common reference string. Therefore even minor changes to the relation require another invocation of a trusted third party to execute the SNARK setup algorithm.

In a sense, the gadget writing can be compared to directly writing "circuit silicon", or constructing ASICs. An alternative approach, implemented in two **libsnark** proof systems, is to construct "silicon" capable of executing any computation of a RAM machine. To this end **libsnark** implements a CPU architecture, called TinyRAM [BCG$^+$13b], tailored to be particularly suitable for zk-SNARK proof systems, and two approaches of verifying TinyRAM programs: universal circuits using routing, and recursive proof composition with memory delegation.

Universal circuits for RAM programs. To verify execution of any RAM computation, **libsnark** follows the blueprint laid out in [BCGT13, BCTV14c], and constructs a circuit capable of verifying any RAM computation that runs for no more than T steps. Thus the trusted zk-SNARK generator depends only on this time bound T, but not on the computation itself.

The universal circuit consists of three parts. First, to verify correct execution of non-memory operations performed by the RAM CPU, the circuit has wires encoding the entire execution trace (that is, the CPU state, including program counter, registers, flags, etc at time $t = 0, 1, \dots$), and $T - 1$ subcircuits verifying successive CPU state transition between these T states. Second, the circuit also has wires to encode the entire memory access trace. Each entry in the memory trace includes the time step t, memory address addr, indication whether this address was read from or written to, and the contents of this address after this operation. Unlike the CPU trace, which is sorted by time, the memory trace is sorted first by the address, with ties broken by time. Given such trace it is easy to write $t - 1$ subcircuits that check that consecutive access to the same memory location yield consistent results (e.g. after write instruction the next read instruction returns the same value; values don't change between read instructions; etc). Finally, to ensure the consistency between time and memory traces, the universal circuit also includes a routing network that checks that the two traces are permutations of each other.

TinyRAM. The above approach yields a universal circuit of size $O(T \log T + cT)$, where the constant c depends on the complexity of checking the CPU transition function. The constant for the $O(T \log T)$ term is just 2, therefore in practice the $O(cT)$ term dominates, and to minimize this c, **libsnark** implements a RISC CPU architecture that is tailored for fast verification. As opposed to architectures like Intel x86 that are tailored for fast execution, and thus benefit from highly expressive instruction sets and their extensions, TinyRAM has only 16 instruction types, and can be implemented in around 1000 bilinear gates. We refer to [BCG$^+$13a] for further details and performance comparison.

RAM program verification from recursive proof composition. Universal circuit approach has an inherent scalability drawback: the SNARK prover needs to produce the wire for this universal circuit, encoding the entire computation, and then perform global operations, such as FFTs, on this assignment. To avoid "carrying around" the entire computation trace, **libsnark** uses recursive proof composition [Val08, CT10] and memory delegation techniques to obtain proof system where setup algorithm is universal, yet does not depend on the time bound T, and prover only needs memory proportional to space complexity of the original RAM computation.

In a nutshell, this construction, presented in [BCTV14a] works as follows. Instead of maintaining the contents of memory in a memory trace inside the circuit, we will outsource handling of the memory to the (untrusted) prover and enforce the correctness using Merkle trees. More precisely, we construct a circuit C consisting of three parts: (a) a CPU checker, capable of verifying one RAM machine step; (b) a Merkle tree checker, capable of verifying correctness of one read or store instruction; and (c) a SNARK verifier, for recursively verifying this computation. At each step of the RAM computation, the SNARK prover convinces C that: computation up to this point was executed correctly (i.e. there exists a SNARK proof for this statement), the current RAM step is executed correctly, and that the only change in the Merkle tree is one requested by the RAM CPU.

2.3.3 Case study: an arithmetic circuit for verifying SHA-256's compression function

The vast majority of work in Zerocash's NP statement is verifying computations of $\mathcal{H}$, the compression function of SHA-256. In this Section we briefly contrast the approaches of obtaining efficient circuit implementation for $\mathcal{H}$.

We wish to construct an arithmetic circuit C_{SHA256} such that, for every 256-bit digest h and 512-bit input x, $(h, x) \in \mathcal{R}_{C_{\mathsf{SHA256}}}$ if and only if $h = \mathcal{H}(x)$. Naturally, our goal is to minimize the size of C_{SHA256}. Our high-level strategy is to construct C_{SHA256}, piece by piece, by closely following the SHA-256 official specification [Nat12]. [3] For each subcomputation of SHA-256, we use nondeterminism and field operations to verify the subcomputation using as few gates as possible.

Overview of SHA-256's compression function. The primitive unit in SHA-256 is a 32-bit *word*. All subcomputations are simple word operations: three bitwise operations (and,

[3] The official specification matches common practical SHA256 implementations. Our intuition that as a cryptographic heuristic there should not be "shortcuts" in the hash function's internal structure.

`or`, `xor`), shift-right, rotate-right, and addition modulo 2^{32}. The compression function internally has a *state* of 8 words, initialized to a fixed value, and then transformed in 64 successive rounds by following the 64-word *message schedule* (deduced from the input x). The 256-bit output is the concatenation of the 8 words of the final state.

Representing a state. We find that, for each word operation (except for addition modulo 2^{32}), it is more efficient to verify the operation when its inputs are represented as separate wires, each carrying a bit. Thus, C_{SHA256} maintains the 8-word state as 256 individual wires, and the 64-word message schedule as $64 \cdot 32$ wires.

Addition modulo 32. To verify addition modulo 2^{32} we use techniques employed in previous work [PGHR13, BCG$^+$13a, BCTV14c]. Given two words A and B, we compute $\alpha := \sum_{i=0}^{31} 2^i(A_i + B_i)$. Because $\mathbb{F}$ has characteristic larger than 2^{33}, there is no wrap around; thus, field addition coincides with integer addition. We then make a non-deterministic guess for the 33 bits α_i of α (including carry), and enforce consistency by requiring that $\alpha = \sum_{i=0}^{32} 2^i\alpha_i$. To ensure that each $\alpha_i \in \{0, 1\}$, we use a 33-gate subcircuit computing $\alpha_i(\alpha_i - 1)$, all of which must be 0 for the subcircuit to be satisfiable. Because our finite field has high characteristic, the fact that these checks are satisfied modulo p, mean that they also pass over integers as well. Overall, verifying addition modulo 2^{32} only requires 34 gates. This approach extends in a straightforward way to summation of more than two terms.

Verifying the `SHA-256` message schedule. The first 16 words W_i of the message schedule are the 16 words of the 512-bit input x. The remaining 48 words are computed as $W_t := \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$, where $\sigma_0(W) := \mathtt{rotr}_7(W) \oplus \mathtt{rotr}_{18}(W) \oplus \mathtt{shr}_3(W)$ and σ_1 has the same structure but different rotation and shift constants.

The rotation and shift amounts are constants, so rotates and shifts can be achieved by suitable wiring to previously computed bits (or the constant 0 for high-order bits in `shr`). Thus, since the XOR of 3 bits can be computed using 2 gates, both σ_0 and σ_1 can be computed in 64 gates. We then compute (or more precisely, guess and verify) the addition modulo 2^{32} of the four terms.

Verifying the `SHA-256` round function. The round function modifies the 8-word state by changing two of its words and then permuting the 8-word result. Each of the two modified words is a sum modulo 2^{32} of (i) round-specific constant words K_t; (ii) message schedule words W_t; and (iii) words obtained by applying simple functions to state words. Two of those functions are bitwise *majority* ($\mathsf{Maj}(A, B, C)_i = 0$ if $A_i + B_i + C_i \leq 1$ else 1) and bitwise *choice* ($\mathsf{Ch}(A, B, C)_i = B_i$ if $A_i = 1$, else C_i). We verify correct computation of Maj

using 2 gates per output bit, and Ch with 1. Then, instead of copying 6 unchanged state words to obtain the permuted result, we make the permutation implicit in the circuit's wiring, by using output wires of previous sub-computations (sometimes reaching 4 round functions back) as input wires to the current sub-computation.

Performance. Overall, we obtain an arithmetic circuit C_{SHA256} for verifying SHA-256's compression function with less than 30 000 arithmetic gates. See Figure 2.2 for a breakdown of gate counts.

Gate count for C_{SHA256}	
Message schedule	8 032
All rounds	19 584
1 round (of 64)	306
Finalize	288
Total	27 904

Figure 2.2: Size of circuit C_{SHA256} for SHA-256's compression function.

Comparison with generic approaches. We constructed the circuit C_{SHA256} from scratch. We could have instead opted for more generic approaches: implement SHA-256's compression function in a higher-level language, and use a circuit generator to obtain a corresponding circuit. However, generic approaches are significantly more expensive for our application, as we now explain.

Starting from the SHA-256 implementation in PolarSSL (a popular cryptographic library) [Pol13], it is fairly straightforward to write a C program for computing $\mathcal{H}$. We wrote such a program, and gave it as input to the circuit generator of [PGHR13]. The output circuit had 58160 gates, more than twice larger than our hand-optimized circuit.

Alternatively, we also compiled the same C program to TinyRAM, which is the architecture supported in [BCG$^+$13a]; we obtained a 5371-instruction assembly code that takes 5704 cycles to execute on TinyRAM. We could then invoke the circuit generator in [BCG$^+$13a] when given this TinyRAM program and time bound. However, each TinyRAM cycle costs $\approx$ 1000 gates, so the resulting circuit would have at least $5.7 \cdot 10^6$ gates, i.e., over 190 times larger than our circuit.1 A similar computation holds for the circuit generator in [BCTV14c], which supports an even more flexible architecture.

Thus, overall, we are indeed much better off constructing C_{SHA256} from scratch. Of course, this is not surprising, because a SHA-256 computation is almost a "circuit computation": it does not make use of complex program flow, accesses to memory, and so on. Thus, relying on machinery developed to support much richer classes of programs does not pay off.

2.4 Shared algorithmic core in libsnark

2.4.1 Finite field arithmetic

Cryptographic encodings in **libsnark** are based on pairing-friendly elliptic curves over finite fields. Consequently, essentially all cryptographic work performed by the library relies on fast finite field arithmetic. Moreover, the largest non-cryptographic work done by the zk-SNARK prover involves polynomial interpolation and evaluation, again over finite fields, and, in fact, operations that do not translate to primitive finite-field operations contributes negligibly to the runtimes of zk-SNARK generator, prover, and verifier.

Typical high performance elliptic curve implementations target curves over fields $\mathbb{F}_q$, where q is specifically chosen to either have low Hamming weight or be close to power of two; this choice is made so that the finite field multiplication could benefit from efficient modular reduction algorithms that only work for primes of this structure. Unfortunately, these optimizations cannot be applied to the zk-SNARK setting: to enable fast polynomial interpolation and evaluation, we chose our curves in such a way that r, the order of the curve, has property that $r-1$ is divisible by a large power of two. That way, $\mathbb{F}_r$ has necessary 2^k-th roots of unity, and one can apply particularly efficient radix-2 FFTs.

Therefore, to speed up the finite field multiplication, **libsnark** uses the Montgomery representation [Mon85] and the Coarsely Integrated Operand Scanning (CIOS) method [KAK96] for simultaneous multiplication with modular reduction. The finite field routines in **libsnark** are written in optimized assembly, but the library also includes generic C++ implementations of the field arithmetic, mainly for portability and auditing reasons.

2.4.2 Bilinear group arithmetic based on elliptic curves

We assume familiarity with elliptic curves; here, we only recall the basic definitions in order to fix notation. See, e.g., [Was08, Sil09, FST10, CFA+12] for more details.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be cyclic groups of a prime order r. We denote elements of $\mathbb{G}_1, \mathbb{G}_2$ via calligraphic letters such as $\mathcal{P}, \mathcal{Q}$. We write $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation. Let $\mathcal{P}_1$ be a generator of $\mathbb{G}_1$, i.e., $\mathbb{G}_1 = \{\alpha\mathcal{P}_1\}_{\alpha \in \mathbb{F}_r}$; let $\mathcal{P}_2$ be a generator for $\mathbb{G}_2$. (We also view α as an integer, so that $\alpha\mathcal{P}_1$ is well-defined.) A **pairing** is an efficient map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_T$ is also a cyclic group of order r (which we write in multiplicative notation), satisfying the following properties:

- BILINEARITY. For every nonzero elements $\alpha, \beta \in \mathbb{F}_r$, it holds that $e(\alpha\mathcal{P}_1, \beta\mathcal{P}_2) =$

$e(\mathcal{P}_1, \mathcal{P}_2)^{\alpha\beta}$.

- NON-DEGENERACY. $e(\mathcal{P}_1, \mathcal{P}_2)$ is not the identity in $\mathbb{G}_T$.

A pairing is typically instantiated via a *pairing-friendly elliptic curve*. Concretely, suppose that one uses a curve E defined over $\mathbb{F}_q$, with embedding degree k with respect to r, to instantiate the pairing. Then $\mathbb{G}_T$ is set to μ_r, the subgroup of r-th roots of unity in $\mathbb{F}^*_{q^k}$. The instantiation of $\mathbb{G}_1$ and $\mathbb{G}_2$ depends on the choice of e; typically, $\mathbb{G}_1$ is instantiated as an order-r subgroup of $E(\mathbb{F}_q)$, while, for efficiency reasons [BKLS02, BLS04], $\mathbb{G}_2$ as an order-r subgroup of $E'(\mathbb{F}_{q^{k/d}})$ where E' is a *d-th twist* of E. Finally, the pairing e is typically a two-stage function $e(\mathcal{P}, \mathcal{Q}) := \mathsf{FE}(\mathsf{ML}(\mathcal{P}, \mathcal{Q}))$, where $\mathsf{ML}\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_q^k$ is known as *Miller loop*, and $\mathsf{FE}\colon \mathbb{F}_q^k \to \mathbb{F}_q^k$ is known as *final exponentiation* and maps α to $\mathsf{FE}(\alpha) := \alpha^{(q^k-1)/r}$.

When only making "black-box" use of a pairing, the typical backend SNARK verifier must evaluate 12 pairings, amounting to 12 Miller loops plus 12 final exponentiations. The straightforward approach is to compute these using a generic high-performance pairing library. The implementations in **libsnark** proceed differently: we obtain high-performance implementations of *sub-components* of a pairing, and then tailor their use specifically to V's protocol. Namely, first, we first obtain implementations of a Miller loop and final exponentiation.

The SNARK verifier implementations in **libsnark** leverage the fact that a product of pairings can be evaluated faster than evaluating each pairing separately and then multiplying the results [Sol03, Sco05, GS06, Sco07]. Concretely, in a product of m pairings, the Miller loop iterations for evaluating each factor can be carried out in "lock-step" so to share a single *Miller accumulator variable*, using one $\mathbb{F}_{q^k}$ squaring per loop instead of m. In a similar vein, one can perform a single final exponentiation on the product of the outputs of the m Miller loops, instead of m final exponentiations and then multiplying the results. In fact, since the output of the pairing can be inverted for free (as the element is *unitary* so that inverting equals conjugating [SB04]), the idea of "sharing" final exponentiations extends to a ratio of pairing products. Concretely, for [PGHR13] verifier we only need to perform 5, instead of 12, final exponentiations.

2.4.3 Fixed and variable base multiexponentiation

Variable-base multiexponentiation. The SNARK prover P faces several large instances of a *multi-exponentiation problem*, a well-studied computational problem in applied cryptography [Ber02]. The problem is as follows: given group elements $g_1, \ldots, g_m \in \mathbb{G}$ (here, $\mathbb{G} = \mathbb{G}_1$ or $\mathbb{G} = \mathbb{G}_2$) and 256-bit integers $a_1, \ldots, a_m$, compute $\prod_{i=1}^{m} g_i^{a_i}$. In order to reduce the number of group operations required to compute this product, we **libsnark** implements two multi-exponentiation algorithm [BC89] and [BDLO12], with different trade-offs. Compared to the naive approach of "exponentiate and then multiply", this save a multiplicative factor of between 25 (for [BC89]) and 35 (for [BDLO12]) already for $m = 10^6$ (and the savings increase with m).

Fixed-base multiexponentiation. The SNARK generator G is instead faced with several large instances of the following exponentiation problem: given a group element $g \in \mathbb{G}$ and 256-bit integers $a_1, \ldots, a_m$, compute the tuple $(g^{a_1}, \ldots, g^{a_m})$. **libsnark** reduce the number of required group operations by using the standard technique of pre-computing a table of powers of g, and then reusing these values in each subsequent exponentiation. For 256-bit exponents, this saves a multiplicative factor of 23 in the number of group operations (over the naive approach of performing a "fresh" exponentiation for each term). Precomputing more powers of g provides even greater savings, at the expense of more space usage.

2.4.4 Routing network algorithms

For fast RAM reductions **libsnark** implements two routing networks: a Beneš networks, and an *arbitrary-size Waksman networks* [BÉ02]. The latter requires $N(\log N - 0.91)$ switches to route N packets, instead of $2^{\lceil \log N \rceil}(\lceil \log N \rceil - 0.5)$ for the former. Besides being closer to the information-theoretic lower bound of $N(\log N - 1.443)$, such networks eliminate costly rounding effects in [BCG+13a], where the size of the network is *doubled* if N is just above a power of 2 (since the height of a Beneš network is $2^{\lceil \log N \rceil}$). That said, Beneš are conceptually much simpler, which helps with parallel routing implementations (see Section 2.5).

2.5 Impact of libsnark and the future of the library

Future optimizations. Essentially all the computations required by the proof system backend's generator, prover, and verifier can be parallelized. In particular, routing on Beneš networks, sorting, polynomial interpolation/evaluation, multi-exponentiation, and others — all of these are highly-parallelizable (i.e., have polylogarithmic-depth circuits). Parallel implementations of all of these computational tasks are well-studied, and it should not be difficult to make **libsnark** leverage all available cores so to significantly reduce latency. We currently only do so for the two "heavy hitters": batch multiexponentiation routines and,to a limited extent, FFTs.

Chapter 3

Secure sampling of public parameters for succinct zero knowledge proofs

3.1 Introduction

Non-interactive zero-knowledge proof systems require a trusted party to sample and publish a set of public parameters; subsequently, anyone can use the public parameters to produce and verify proofs. In this Chapter, we design, build, and evaluate a multi-party protocol for securely sampling public parameters of a class of non-interactive zero-knowledge proof systems. Informally, if n parties participate in the protocol and at least one of them is honest, then (i) the protocol's output consists of public parameters sampled from the correct distribution; and, (ii) the protocol's transcript leaks no information beyond the public parameters themselves. The class of proof systems supported by our protocol includes state-of-the-art constructions with short and easy-to-verify proofs [PGHR13, BCTV14c, DFGK14] and, for these, achieves excellent concrete efficiency. For example, our protocol can efficiently generate public parameters for Zerocash [BCG$^+$14] and for the scalable zero-knowledge proof system of [BCTV14b].

3.1.1 Motivation

In recent years individuals and enterprises have begun to migrate large quantities of internal data to outside providers. This trend raises concerns about the integrity and confidentiality of computations conducted on this data. Consider, for example, the following simple illustrative scenario. A server owns a private database x, and a client wishes to learn $y := F(x)$ for a public function F; a commitment cm to x is known publicly. For example, x may be a database of genetic data, and F may be a machine-learning algorithm that uses the genetic data to compute a classifier y. On the one hand, the client seeks *integrity of computation*: he wants to ensure that the server reports the correct output y (e.g., because the classifier y will be used for critical medical decisions). On the other hand, the server seeks *confidentiality* of his own input: he is willing to disclose y to the client, but no additional information about x beyond y (e.g., because the genetic data x may contain sensitive personal information).

Zero-knowledge proofs. Achieving the combination of the above security requirements seems paradoxical: the client does not have the input x, and the server is not willing to share it. Yet, cryptography offers a powerful tool that can do that: *zero-knowledge proofs* [GMR89, GMW91]. The server, acting as the prover, attempts to convince the client, acting as the verifier, that the following NP statement is true: "there is $\tilde{x}$ such that $y = F(\tilde{x})$ and $\tilde{x}$ is a decommitment of cm". Indeed: (a) the proof system's *soundness* property addresses the client's integrity concern, because it guarantees that, if the NP statement is false, the prover cannot convince the verifier (with high probability);[1] and (b) the proof system's *zero-knowledge* property addresses the server's confidentiality concern, because it guarantees that, if the NP statement is true, the prover can convince the verifier without leaking any information about x (beyond what is leaked by y).

Non-interactivity. While zero-knowledge proofs can address the above simple scenario, they also apply more broadly, including to scenarios that involve many parties who do not trust each other or are not all simultaneously online. In such cases, it is desirable to use *non-interactive zero-knowledge proofs* (NIZKs), where the proof consists of a single message π that can be verified by anyone. For example, a non-interactive proof π can be stored for later use, or it can be verified by multiple parties without requiring the prover to separately interact with each one of these.

Unfortunately, NIZKs do not exist for languages outside BPP (even when soundness is relaxed to hold only computationally) [GO94]. But, if a trusted party is available for a one-time setup phase, then, under suitable hardness assumptions, NIZKs exist for all languages in NP [BFM88, BDSMP91, FLS99]. During the setup phase, the trusted party runs a probabilistic polynomial-time *generator* algorithm G (prescribed by the proof system) and publishes its output pp, called the *public parameters* (or *common reference string*);

[1] Sometimes a property stronger than soundness is required: *proof of knowledge* [GMR89, BG93], which guarantees that, whenever the verifier is convinced, not only can he deduce that a witness exists, but also that the prover *knows* one such witness.

afterwards, the trusted party is no longer needed, and anyone can use pp to produce proofs or to verify them.

Soundness of the NIZK depends on this trusted setup: if pp is not correctly generated, or if secret internal randomness used within G is revealed, then it may be feasible to convince the verifier of false NP statements. Compromised soundness can have catastrophic implications, because an attacker can cause significant damage without being detected.

The problem of parameter generation. If no trusted party is available, how should the public parameters pp be generated? Without *some* trustworthy method to generate public parameters, deploying practical systems that rely on NIZKs (e.g., Zerocash [BCG$^+$14]) seems very challenging.

First of all, if the public parameters are "simple", then generating them securely may be easier. For instance, a notable special case is when pp is a uniformly random binary string of a certain length (known as a *common random string*). In this case, several approaches could be investigated: (i) Utilize coin-tossing protocols (ii) Look for, in Nature or Society, a publicly-observable distribution that equals, or is close to, random: via suitable measurements and post-processing of, e.g., data about sun spots or the stock market, it may be possible to extract bits that are close to random (see [CPS07, CH10] for work in this direction, and [Nat14] for a NIST prototype using quantum randomness sources). (iii) Resort to heuristics that use deterministic yet "random-looking" bits, e.g., consider an agreed-upon block of bits from the mathematical constant π, or the output of SHA-256 on an agreed-upon input.

However, if G generates pp by following a complex probabilistic strategy, then the above approaches may not apply.

Distributed parameter generation. An attractive approach to address the problem of parameter generation is to design a multi-party protocol for securely sampling pp. That is, the setup phase involves a set of parties running the multi-party protocol for generating pp, and for soundness of the NIZK to hold it suffices that only a few (ideally, even just one) of these parties are honest. Clearly, relying on such a distributed protocol is a weaker and more realistic trust assumption than placing ultimate trust in any single party.

Several works have explored this approach for generating public parameters of various cryptographic primitives and, more generally, one can utilize secure multi-party computation [GMW87b, BOGW88] to obtain a feasibility result. Yet, as discussed in Section 3.2, prior work does not yield satisfactory efficiency in our setting, which we now introduce.

3.1.2 Our focus

The problem of parameter generation has garnered recent attention due to the development of new and powerful NIZKs that enable verifying general computation via proofs that are *succinct*, i.e., short and easy to verify [Mic00]. The new proofs are known as *zero-knowledge succinct arguments of knowledge* (zk-SNARKs) [GW11, BCCT12, BCI$^+$13], and have already found practical applications, e.g., to building privacy-preserving decentralized electronic cash [BCG$^+$14]. Most zk-SNARKs require an involved parameter generation, often with complexity proportional to the size of the computation being proved; addressing this parameter generation is the focus of our work. Concretely, we obtain efficient multi-party protocols for securely sampling the public parameters required by zk-SNARKs, as we now explain.

zk-SNARK constructions. There are many zk-SNARK constructions, with different properties in efficiency and supported languages. In *preprocessing zk-SNARKs*, the complexity of sampling public parameters grows with the size of the computation being proved [Gro10, Lip12, BCI$^+$13, GGPR13, PGHR13, BCG$^+$13a, Lip13, FLZ13, BCTV14c, Lip14, KPP$^+$14, ZPK14, DFGK14, WSR$^+$15, BBFR15]; in *fully-succinct zk-SNARKs*, that complexity is independent of computation size [Mic00, Val08, Mie08, DL08, BCCT12, DFH12, GLR11, BC12, BCCT13, BCTV14b, BCC$^+$14]; yet other constructions strike tradeoffs between these two extremes [CTV15, CFH$^+$15]. Working prototypes have been achieved for preprocessing zk-SNARKs [PGHR13, BCG$^+$13a, BCTV14c, KPP$^+$14, ZPK14, BBFR15], fully-succinct ones [BCTV14b], and other kinds [CTV15, CFH$^+$15]. Several works have also explored various applications of zk-SNARKs [CKLM13, BFR$^+$13, DFKP13, BCG$^+$14, FL14].

Public parameters of zk-SNARKs. Despite the aforementioned multitude of constructions, Bitansky et al. [BCI$^+$13] showed that essentially all known preprocessing zk-SNARK constructions can be "explained" as the combination of a *linear interactive proof* (LIP) and a cryptographic encoding that only supports linear homomorphisms. This yields a unified view of parameter generation across preprocessing zk-SNARKs (that are not fully succinct). Namely, given an NP relation $\mathcal{R}$, the generator G adheres to the following computation pattern when producing public parameters for $\mathcal{R}$: (i) derive from $\mathcal{R}$ a certain circuit C (essentially, C is the multi-output circuit that computes the LIP's verifier's message); (ii) evaluate C at a random input; (iii) output the encoding of the evaluation. In other words, public parameters of preprocessing zk-SNARKs are the encodings of random evaluations of certain circuits.

The sampling problem. More precisely, the zk-SNARK fixes a prime r and an order-r

group $\mathbb{G}$ with generator $\mathcal{G}$; then, a derived circuit C is defined over the field $\mathbb{F}_r$, and the encoding of an element α in $\mathbb{F}_r$ is $\alpha \cdot \mathcal{G}$ (where we view α as an integer to define its group action); when $\vec{a} = (a_1, \ldots, a_n)$, we use $\vec{a} \cdot \mathcal{P}$ as a shorthand for the vector $(a_1 \cdot \mathcal{P}, \ldots, a_n \cdot \mathcal{P})$. This discussion motivates the following multi-party sampling problem:

> Let r be a prime, $\mathbb{G} = \langle \mathcal{G} \rangle$ an order-r group, n a positive integer, and $C \colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ an $\mathbb{F}_r$-arithmetic circuit. Construct an n-party protocol for securely sampling $\vec{\mathcal{P}} := C(\vec{\alpha}) \cdot \mathcal{G}$, lying in $\mathbb{G}^h$, for a random $\vec{\alpha}$ in $\mathbb{F}^m$.

We thus seek a multi-party protocol such that, even when all but one of the n parties are malicious, the protocol's output is $\vec{\mathcal{P}}$ sampled from the correct distribution (or at least from one indistinguishable from it) and, moreover, the n parties, as well as any others observing the protocol's execution, learn nothing beyond $\vec{\mathcal{P}}$ itself. We study this problem, and the special case of generating public parameters for preprocessing zk-SNARKs.

3.1.3 Our contributions

We design, build, and evaluate a multi-party protocol for securely sampling encodings of random evaluations of certain circuits.

The resulting system enables us, in particular, to sample the public parameters for a class of preprocessing zk-SNARKs that includes [PGHR13, BCTV14c, DFGK14]; we integrated our system with **libsnark** [SCI], a C++ zk-SNARK library, to facilitate this application. In more detail, we present the following two main contributions.

(1) Secure sampling for a class of circuits. We design, build, and evaluate a multi-party protocol that securely samples values of the form $C(\vec{\alpha}) \cdot \mathcal{G}$ for a random $\vec{\alpha}$, provided that C belongs to a certain circuit class $\mathbf{C}^{\mathsf{S}}$. Roughly, $\mathbf{C}^{\mathsf{S}}$ comprises the $\mathbb{F}$-arithmetic circuits for which: (i) the output of each (addition or multiplication) gate is an output of the circuit; (ii) the inputs of each addition gate are outputs of the circuit; (iii) the two inputs of each multiplication gate are, respectively, a circuit output and a circuit input or an output of an addition-free subcircuit. See Figure 3.1a on page 46 for an example of a circuit in $\mathbf{C}^{\mathsf{S}}$.

The multi-party protocol is based on standard cryptographic assumptions, and runs atop a synchronous network with an authenticated broadcast channel and a common random string. The computation proceeds in rounds and, at each round, the protocol's schedule determines which parties act; a party acts by broadcasting a message to all others.

When n parties participate, our protocol is secure against up to $n-1$ non-adaptive corruptions. If even one of the parties is honest, and assuming the protocol reaches completion, then the protocol's output is a sample from the designated distribution and no other information leaks.[2] Each party runs in time $O_\lambda(\mathsf{size}(C))$, where $O_\lambda(\cdot)$ hides a fixed polynomial in the security parameter λ. The number of rounds is $n \cdot \mathsf{depth}_\mathsf{S}(C) + O(1)$ and the number of broadcast messages is $O(n \cdot \mathsf{depth}_\mathsf{S}(C))$. Here, $\mathsf{depth}_\mathsf{S}(C)$ denotes the S-*depth* of C (introduced later), which is at most the standard circuit depth of C, but sometimes much smaller, as is the case for the zk-SNARK application discussed below.

While the above results hold for any group $\mathbb{G}$, our code implementation is specific to *duplex-pairing* groups, i.e., $\mathbb{G}$ is a subgroup of some $\mathbb{G}_1 \times \mathbb{G}_2$ equipped with a pairing. This holds in the zk-SNARK application, and enables further optimizations (for which we additionally rely on random oracles so as to benefit from the Fiat–Shamir heuristic [FS87]).

Compared to previous results in secure multi-party computation protocols, our specialized construction scales up to larger number of participating parties without incurring a high round complexity; see Section 3.2.

(2) Application to zk-SNARKs. Our protocol can securely sample public parameters of a zk-SNARK, whenever the generator can be cast as sampling the encoding of the random evaluation of a circuit in the class $\mathbf{C}^\mathsf{S}$.

While $\mathbf{C}^\mathsf{S}$ is a restrictive class, we show that several preprocessing zk-SNARK constructions have such a generator:

- **zk-SNARKs for arithmetic relations.** For any arithmetic circuit D, we obtain a circuit C in $\mathbf{C}^\mathsf{S}$ such that the encoding of a random evaluation of C corresponds to public parameters for [PGHR13, BCTV14c]'s zk-SNARK when proving D's satisfiability. The circuit C has size $O(\mathsf{size}(D) \log \mathsf{size}(D))$ and S-depth $O(1)$.

- **zk-SNARKs for boolean relations.** For any boolean circuit D, we obtain a circuit C in $\mathbf{C}^\mathsf{S}$ such that the encoding of a random evaluation of C corresponds to public parameters for [DFGK14]'s zk-SNARK when proving D's satisfiability. The circuit C has size $O(\mathsf{size}(D) \log \mathsf{size}(D))$ and S-depth $O(1)$.

To facilitate the application to zk-SNARKs we integrated our multi-party protocol, as well as the aforementioned circuit transformations, with **libsnark** [SCI]. Along the way, we also extended **libsnark** with an implementation of [DFGK14]'s zk-SNARK, augmenting its existing implementation based on [PGHR13, BCTV14c].

[2] A malicious party may prevent the protocol from completing, by acting incorrectly or by delaying prescribed broadcasts. However, the culprit can be readily identified. Such aborts necessarily bias the output distribution but, in our case, the bias is negligible and thus inconsequential.

We evaluated our protocol's costs when invoked to securely sample the public parameters of the zk-SNARK used in the following two concrete examples.

- **Example #1:** *Zerocash* (a decentralized anonymous payment system extending Bitcoin) [BCG$^+$14]. We fix D to be the arithmetic circuit used in [BCG$^+$14]'s zk-SNARK. Then C has size 138 467 206 and S-depth 3; in our multi-party protocol, the number of rounds is $3n + 3$ and, when counting cryptographic work on our reference system, each party works for 14 124 s.
- **Example #2:** *scalable ZK* (incrementally-computable zero knowledge for a 32-bit RISC architecture) [BCTV14b].

  We fix D to be the arithmetic circuit used in [BCTV14b]'s zk-SNARK. Then C has size 8 027 609 and S-depth 6; in our multi-party protocol, the number of rounds is $6n + 6$ and each party works for 4 048 s. (In [BCTV14b] there are *two* required circuits D; here and later we specify, for each complexity measure, the sum of the two costs.)

In both cases above, C's S-depth is less than 10, while its standard depth exceeds many hundreds of thousands. The fact that our sampling protocol's round complexity is efficient in S-depth allows for scaling to larger number of parties.

Remark 3.1.1. We stated that our multi-party protocol runs atop a synchronous network with an authenticated broadcast channel and a common random string. In practice, one needs to realize (or at least approximate) this model. A careful exploration of which realizations are suitable lies outside the scope of this work, and is anyways application dependent. We only briefly mention natural options to consider: a broadcast channel can be viewed as an append-only public logbook and may be obtained, e.g., via Bitcoin's Proof-of-Work-based blockchain protocol [Nak09]; authentication may be obtained, e.g., via digital signatures supported by a public-key infrastructure; a common random string may be obtained, e.g., via a public randomness source with high entropy or coin-tossing protocols.

3.1.4 Summary of challenges and techniques

We describe at a high level the challenges that arise, as well as the techniques that we employed to address them, for each of our two main contributions.

Secure sampling for a class of circuits

Let r be a prime, $\mathbb{G} = \langle \mathcal{G} \rangle$ an order-r group, n a positive integer, and $C \colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ an $\mathbb{F}_r$-arithmetic circuit. We seek an n-party protocol for sampling $C(\vec{\alpha}) \cdot \mathcal{G}$, for a random $\vec{\alpha}$, that is secure against up to $n-1$ non-adaptive corruptions. We may compromise on functionality by restricting C to lie in a circuit class $\mathbf{C}^{\mathsf{S}}$, provided that we gain improved efficiency (since, ultimately, we want to implement the protocol and use it to generate zk-SNARK public parameters).

The ideal functionality. We first need to choose the ideal functionality $f_{n,C,\mathcal{G}}$ to be realized by the multi-party protocol. A reasonable candidate is the following: on input $\vec{\sigma} := (\vec{\sigma}_1, \dots, \vec{\sigma}_n)$ where $\vec{\sigma}_i = (\sigma_{i,1}, \dots, \sigma_{i,m}) \in \mathbb{F}_r^m$ is the i-th party's input, $f_{n,C,\mathcal{G}}$ first computes $\alpha_j := \prod_{i=1}^n \sigma_{i,j}$ for $j = 1, \dots, m$; then $f_{n,C,\mathcal{G}}$ sets $\vec{\alpha} := (\alpha_1, \dots, \alpha_m)$ and computes $\vec{\mathcal{P}} := C(\vec{\alpha}) \cdot \mathcal{G}$; finally, $f_{n,C,\mathcal{G}}$ outputs $\vec{\mathcal{P}}$. Indeed, if at least one party honestly provides an input consisting of random field elements, $f_{n,C,\mathcal{G}}$ outputs the encoding of a random evaluation of C.[3]

Difficulties with standard approaches. Realizing an ideal functionality typically (though not always) comprises two steps: (a) express the ideal functionality as a circuit, and then (b) invoke a known multi-party protocol to securely evaluate the circuit. In our setting both steps pose efficiency challenges, which we now discuss.

- *Expressing the ideal functionality as a circuit.*

  Expressing $f_{n,C,\mathcal{G}}$ as a boolean is circuit is expensive, because computing $C(\vec{\alpha}) \cdot \mathcal{G}$ involves "non-boolean" operations: (i) the evaluation of the $\mathbb{F}_r$-arithmetic circuit C, and (ii) h scalar multiplications over the group $\mathbb{G}$. Indeed:

  - The number of boolean gates for evaluating C is at least $\times \log_2 r$ larger than the number of $\mathbb{F}_r$-arithmetic gates for the same task, as each addition and multiplication in $\mathbb{F}_r$ is expanded into a boolean subcircuit of size $\geq \log_2 r$.
  - Each of the h scalar multiplications over $\mathbb{G}$ similarly expands into a boolean subcircuit of size $\geq a_{\mathbb{G}} \cdot \log_2 r$, where $a_{\mathbb{G}}$ is the number of boolean gates to evaluate addition in $\mathbb{G}$.

  In the applications that we consider, $\mathbb{G}$ is instantiated via an elliptic curve over a base field $\mathbb{F}_q$, so that $a_{\mathbb{G}}$ is at least $\log_2 q$. Overall, conversion to boolean operations incurs a

[3] More precisely, $f_{n,C,\mathcal{G}}$ also checks that none of the parties' inputs contains a zero. Forbidding zeros biases the output distribution, but only negligibly, since r is chosen large enough for discrete log to be hard in $\mathbb{Z}_r^*$. The alternative option of additive (rather than multiplicative) shares ultimately results into a construction with worse efficiency.

blowup of up to five orders of magnitude in the number of gates to securely evaluate, because $\log_2 q, \log_2 r \geq 250$ (for, e.g., achieving 128 bits of security for $\mathbb{G}$'s DL problem).

Expressing $f_{n,C,\mathcal{G}}$ as an arithmetic circuit is also expensive, as we now explain. While the circuit C is defined over the field $\mathbb{F}_r$, the group $\mathbb{G}$ may not be; e.g., in our applications, the field $\mathbb{F}_r$ is different from the field $\mathbb{F}_q$ that underlies $\mathbb{G}$[4] . Hence, if we express $f_{n,C,\mathcal{G}}$ as an $\mathbb{F}_r$-arithmetic circuit then, while evaluating C may be efficient, scalar multiplications over $\mathbb{G}$ are not. Conversely, if we express $f_{n,C,\mathcal{G}}$ as an $\mathbb{F}_q$-arithmetic circuit, while scalar multiplications over $\mathbb{G}$ may be efficient, evaluating C is not. Either way, we again incur the overheads associated to mismatch of field characteristic.

- *Securely evaluating the circuit.*

  Known multi-party protocols that are secure against malicious majorities either (i) have round complexity that scales linearly with circuit depth, or (ii) rely on heavy cryptographic tools that are unlikely to yield efficient implementations in the near future. The applications that we consider involve circuits C with depths exceeding hundreds of thousands, so that these prior works do not seem applicable. (See Section 3.2 for discussion and citations.)

Our approach. We take an approach that (a) avoids the overheads due to mismatch in field characteristic, and (b) has low round complexity without relying on heavy cryptographic tools.

We observe (see Section 3.1.4 below) that for several zk-SNARK constructions the "generator circuit" C can, with some effort, be written so as to belong to the circuit class $\mathbf{C}^{\mathsf{S}}$ (informally introduced in Section 3.1.3). We then restrict our attention to realizing the ideal functionality $f_{n,C,\mathcal{G}}$ for $C \in \mathbf{C}^{\mathsf{S}}$.

For circuits in $\mathbf{C}^{\mathsf{S}}$, we design a multi-party protocol in which parties do not follow the two-step recipe of expressing $f_{n,C,\mathcal{G}}$ as a circuit and securely evaluating that circuit (which would likely involve first securely evaluating $\vec{\beta} := C(\vec{\alpha})$ and then securely evaluating its encoding $\vec{\beta} \cdot \mathcal{G}$). Instead, via a suitable use of commitment schemes and NIZKs (in the common random string model), the parties jointly evaluate each gate of C, *directly in encoded form*, eventually producing the final output $C(\vec{\alpha}) \cdot \mathcal{G}$.

At a high level, the construction is as follows. First, as in traditional multi-party protocols, each party commits to his own shares of the input. Then, for each gate of C:

[4] In particular, our applications rely on pairing friendly-curves, where the case $\mathbb{F}_r = \mathbb{F}_q$ would make the discrete logarithm problem easy [MOV91].

(i) if the gate is an addition, each party simultaneously performs the addition locally, by adding the gate's encoded inputs to obtain the gate's encoded output; (ii) if the gate is a multiplication, each party, in sequence, contributes, and proves in zero knowledge correct contribution of, his input shares in producing the encoded output. In both cases, we rely on the fact that the encoding $\alpha \mapsto \alpha \cdot \mathcal{G}$ is linearly homomorphic and that C belongs to $\mathbf{C}^{\mathsf{S}}$. Roughly, these two properties ensure that when a party needs to homomorphically evaluate an addition, its encoded inputs have already been broadcast; similarly, when a party needs to contribute his input shares to one input of a multiplication, the encoding of the other input has already been broadcast.

A naive realization of the above strategy yields an enormous number of broadcast rounds: n times C's depth. In contrast, we show that, via a careful scheduling of when each party processes a multiplication gate, we can reduce the number of broadcast rounds to only n times C's S-depth, where S-depth is a much milder notion of depth (defined later). In the zk-SNARK application that we consider, $\mathsf{depth}(C)$ grows with $\mathsf{size}(C)$ while $\mathsf{depth}_{\mathsf{S}}(C)$ is a small constant.

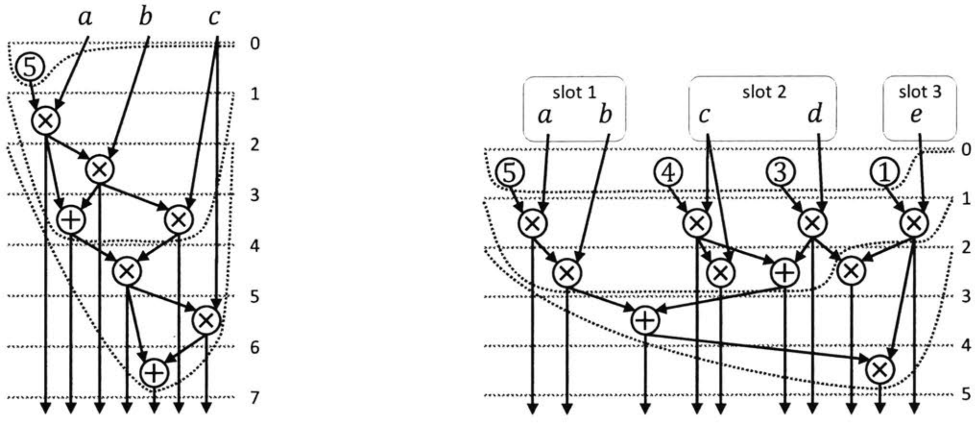
In Section 3.1.5 we sketch in more detail our construction. Our code implementation is specialized to when $\mathbb{G}$ is a duplex-pairing group, in which case the NIZKs used by parties can be implemented very efficiently via Sigma protocols and the Fiat–Shamir heuristic [FS87].

Application to zk-SNARKs

We wish to apply our multi-party protocol to generating public parameters for two zk-SNARK constructions: that of [PGHR13, BCTV14c] and of [DFGK14]. This requires a procedure for transforming the NP relation $\mathcal{R}$ given as input to generator (represented as an instance of circuit satisfiability), into a corresponding circuit $C \in \mathbf{C}^{\mathsf{S}}$ such that the ideal functionality $f_{n,C,\mathcal{G}}$ equals (or is indistinguishable from) the distribution of public parameters output by the generator.

Difficulty with Lagrange polynomials. Constructing the circuit C, subject to the restrictions of $\mathbf{C}^{\mathsf{S}}$ (needed for applying our multi-party protocol), is not straightforward for either of the aforementioned zk-SNARKs. The main technical issue that arises, in both cases, is how to construct an efficient subcircuit of C that evaluates all Lagrange interpolating polynomials at a given input $\tau \in \mathbb{F}_r$.

More precisely, given a subset $W = \{\delta_1, \ldots, \delta_d\}$ of $\mathbb{F}_r$ (potentially enjoying special properties), we seek a circuit $C_W \in \mathbf{C}^{\mathsf{S}}$ of small size and S-depth such that $C_W(\tau) =$



(a) Example of a circuit in $\mathbf{C}^{\mathsf{S}}$.

(b) Example of a circuit in $\mathbf{C}^{\mathsf{E}}$.

Figure 3.1: Examples of a circuit in $\mathbf{C}^{\mathsf{S}}$ and one in $\mathbf{C}^{\mathsf{E}}$. In the latter case, the inputs of the circuit are partitioned into slots. In both cases, the red contour lines denote (traditional) circuit depth, while blue contour lines denote S-depth and E-depth respectively. (See Section 3.1.5 for more details.)

$(L_1(\tau), \ldots, L_d(\tau))$, where $L_i(z)$ denotes the univariate polynomial $\prod_{j=1}^{d}(z - \delta_j) / \prod_{j \neq i}(\delta_j - \delta_i)$.

The simplest setting for interpolation problems typically occurs when d is a power of 2 and W is a subgroup of $\mathbb{F}_r^*$, and thus we focus on it. One can show that, in this case, there is a linear-time algorithm for evaluating all Lagrange polynomials. Unfortunately, this algorithm crucially makes use of division gates, which are not allowed for circuits in $\mathbf{C}^{\mathsf{S}}$ (since our protocol does not handle them). Falling back on naive approaches to evaluate all Lagrange polynomials does yield a circuit in $\mathbf{C}^{\mathsf{S}}$, but of quadratic size and linear depth and S-depth.

Our approach. We provide two constructions of a Lagrange evaluation circuit C_W, with different tradeoffs.

- The first construction has size $O(d \log d)$ and S-depth $O(1)$.
- The second construction has size $O(d)$ and S-depth $O(\log d)$.

In both cases, we rely on a suitable FFT-like subcircuits that lie in $\mathbf{C}^{\mathsf{S}}$.

3.1.5 Construction summary

We summarize our construction of an n-party protocol realizing the ideal functionality $f_{n,C,\mathcal{G}}$ against up to $n-1$ non-adaptive corruptions. We only consider the case when the circuit $C \colon \mathbb{F}^m \to \mathbb{F}^h$ lies in the class $\mathbf{C}^{\mathsf{S}}$, which consists of circuits for which: (i) the output of each (addition or multiplication) gate is an output of the circuit; (ii) the inputs of each addition gate are outputs of the circuit; (iii) the two inputs of each multiplication gate are, respectively, a circuit output and *acircuitinputoranoutputofanaddition − freesubcircuit*. See Figure 3.1a for an example of a circuit in $\mathbf{C}^{\mathsf{S}}$.

We first introduce some ideas for the artificial special case of a single party running the protocol, and then explain how to extend these ideas to multiple parties. Below, $\mathscr{L}_{C,\mathcal{G}}$ denotes the language $\{\vec{\mathcal{Q}} \in \mathbb{G}^h \mid \exists\, \vec{\alpha} \in \mathbb{F}^m \text{ s.t. } \vec{\mathcal{Q}} = C(\vec{\alpha}) \cdot \mathcal{G}\}$, and $\mathcal{D}_{C,\mathcal{G}}$ denotes the distribution over $\mathbb{G}^h$ obtained by drawing $\vec{\alpha} \in \mathbb{F}^m$ at random and then outputting $C(\vec{\alpha}) \cdot \mathcal{G}$.

A special case. Suppose that a single party selects $\vec{\alpha} \in \mathbb{F}^m$ (not necessarily at random), computes the output $\vec{\mathcal{P}} := C(\vec{\alpha}) \cdot \mathcal{G} \in \mathbb{G}^h$, and wishes to broadcast a *transcript* tr that enables anyone to establish correctness of $\vec{\mathcal{P}}$ while not learning any information beyond $\vec{\mathcal{P}}$ itself; we assume that tr contains the claimed output, and denote it by tr.out. More precisely (but still informally), we seek a prover Π, verifier V, and simulator S that satisfy:

(i) *syntactical correctness*: for all efficient Π^*, $\Pr[V(\mathsf{tr}) = 1 \text{ and } \mathsf{tr.out} \notin \mathscr{L}_{C,\mathcal{G}} \mid \mathsf{tr} \leftarrow \Pi^*]$ is negligible; and

(ii) *zero knowledge*: for any $\vec{\mathcal{Q}} \in \mathbb{G}^h$, $S(\vec{\mathcal{Q}})$ is indistinguishable from $\{\mathsf{tr} \mid \mathsf{tr} \leftarrow \Pi\}$ conditioned on $\mathsf{tr.out} = \vec{\mathcal{Q}}$.

At this stage we do not yet pose any requirements on the distribution of the output $\vec{\mathcal{P}}$ claimed in the transcript tr.

One approach to achieve the above is for the single party to broadcast $\mathsf{tr} := (\vec{\mathcal{P}}, \pi)$ where π is a NIZK proof (in the common random string model) for the NP statement "$\vec{\mathcal{P}} \in \mathscr{L}_{C,\mathcal{G}}$", and let V and S be the NIZK verifier and simulator.

While using a NIZK suffices for the case of a single party, we make here several observations that build intuition for the case of multiple parties. First, in many NIZK constructions the proof π is obtained as the concatenation of several sub-proofs attesting to corresponding sub-statements such that, if all of these hold, then the original statement is true. A common paradigm is for the NIZK prover to produce a commitment for each wire in the circuit, and then, for each gate g in C, produce a zero-knowledge sub-proof π_g attesting that the decommitments for the gate's inputs and output are consistent with g. We observe that, if C is in $\mathbf{C}^{\mathsf{S}}$, then the NP statement "$\vec{\mathcal{P}} \in \mathscr{L}_{C,\mathcal{G}}$" factors into a collection of sub-statements of a particular form. Essentially, for each gate g taken in topological order:

- If g is an addition gate, anyone can establish correct evaluation of g by looking up in $\vec{\mathcal{P}}$ the gate's encoded inputs and encoded output (they all appear somewhere in $\vec{\mathcal{P}}$ since all of these are circuit outputs), and check the gate's linear relation by using the encoding's linear homomorphism. So we can set the sub-proof $\pi_g := \perp$.

- If g is a multiplication gate, then we know that the encoding $\mathcal{P}$ of the left input of g and the encoding $\mathcal{R}$ of g's output both appear somewhere in $\vec{\mathcal{P}}$ and, if $\gamma \in \mathbb{F}$ denotes the right input of g, the single party can generate a NIZK sub-proof π_g for the sub-statement "there is γ, consistent with the circuit inputs, such that $\mathcal{R} = \gamma \cdot \mathcal{P}$".

In sum, while we do not know how to enable multiple parties to jointly produce the necessary sub-proofs in the general case, we show that this can be done if C is in $\mathbf{C}^{\mathsf{S}}$, as we discuss next.

Extending to multiple parties. Suppose that there are n participating parties (with $n > 1$), and denote by tr the transcript of all broadcast messages. At a high level, now that there are multiple parties, we seek an n-party protocol Π, verifier V and simulator S that satisfy variants of the above properties:

(i) *distributional correctness*: if at least one party is honest, $\{\mathsf{tr.out} \mid \mathsf{tr} \leftarrow \Pi\}$ conditioned on $V(\mathsf{tr}) = 1$ is indistinguishable from $\mathcal{D}_{C,\mathcal{G}}$; and

(ii) *zero knowledge*: if at least one party is honest, for any $\vec{\mathcal{Q}} \in \mathbb{G}^h$, $S(\vec{\mathcal{Q}})$ is indistinguishable from $\{\mathsf{tr} \mid \mathsf{tr} \leftarrow \Pi\}$ conditioned on $\mathsf{tr.out} = \vec{\mathcal{Q}}$.

Unlike before, now we *are* requiring the output $\vec{\mathcal{P}}$, claimed in the transcript tr, to be distributed in a certain way.

First note that we can't set $\mathsf{tr} := (\mathsf{pp}, \pi)$ where π is a NIZK proof for the NP statement "$\vec{\mathcal{P}} \in \mathscr{L}_{C,\mathcal{G}}$", because there is no single party that knows a witness $\vec{\alpha} \in \mathbb{F}^m$ for it. As specified by the ideal functionality $f_{n,C,\mathcal{G}}$ that we wish to realize (see Section 3.1.4), each party only holds a multiplicative share of every coordinate of $\vec{\alpha}$: party i holds $\vec{\sigma}_i = (\sigma_{i,1}, \ldots, \sigma_{i,m}) \in \mathbb{F}_r^m$ and the j-th coordinate of $\vec{\alpha}$ equals $\prod_{i=1}^n \sigma_{i,j}$.

Yet, the statement "$\vec{\mathcal{P}} \in \mathscr{L}_{C,\mathcal{G}}$" still factors into a collection of sub-statements that, if $C \in \mathbf{C}^{\mathsf{S}}$, are of a particular form. We leverage this fact to design our multi-party protocol. Roughly, for each gate g taken in topological order:

- If g is an addition, similarly to before, we can set the sub-proof π_g to be empty, because anyone can verify the linear relation by looking up in $\vec{\mathcal{P}}$ the encodings of g's inputs and output.
- If g is a multiplication, the sub-statement "there is γ such that $\mathcal{R} = \gamma \cdot \mathcal{Q}$" (mentioned above) further factors into n sub-statements, the i-th one corresponding to contributing the i-th share of γ by party i. Thus, the sub-proof π_g can be jointly computed by concatenating n sub-sub-proofs, the i-th one computed by party i.

Towards a full construction. The construction sketch given so far hides many details, both in terms of defining security goals, formulating the construction, and proving it secure. We briefly describe how we formalize our ideas.

First, instead of targeting the above ad-hoc security properties (introduced for intuition), we formalize our goal as realizing a certain ideal functionality $f_{n,C,\mathcal{G}}$. Our security definitions are thus the standard ones for secure multi-party computation protocols, with the exception that we find it convenient to make explicit the notion of a (transcript) verifier V. See Section 3.4 for more details.

Next, we split our construction in two parts: (1) we reduce the problem of sampling the encoding of a random evaluation of a given circuit C in the class $\mathbf{C}^{\mathsf{S}}$ to the problem of jointly evaluating a related circuit $\tilde{C}$ in another class $\mathbf{C}^{\mathsf{E}}$; (2) we construct a multi-party protocol for securely evaluating any circuit in $\mathbf{C}^{\mathsf{E}}$ (including $\tilde{C}$). More details follow.

The circuit class $\mathbf{C}^{\mathsf{E}}$ differs from $\mathbf{C}^{\mathsf{S}}$ in two ways. First, the inputs of a circuit $\tilde{C} \in \mathbf{C}^{\mathsf{E}}$ are partitioned into *slots*; we write $\tilde{C} \colon \mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n} \to \mathbb{F}^{h}$ to express that the first m_1 inputs are in the first slot, the next m_2 in the second, and so on; the integers $m_1, \ldots, m_n$ are part of $\tilde{C}$'s description. Second, the restriction on the possible inputs of multiplication gates is relaxed (i) the output of each addition gate is an output of the circuit; (ii) the inputs of each addition gate are outputs of the circuit; (iii) the two inputs of each multiplication gate are, respectively, a circuit output and either a circuit input, or a circuit output computable from inputs from a single slot; and (iv) if output of a multiplication gate g is a circuit output, then its left input must be a circuit output and the right input is either a circuit input, or it must be the case that g computes $\mathsf{w}_1 \cdot \frac{\mathsf{w}_2}{\mathsf{w}_3}$ for three circuit output wires $\mathsf{w}_1, \mathsf{w}_2, \mathsf{w}_3$. The last case describes the scenario where a multiplication gate references wires that are not circuit outputs, yet the correctness of this gate can be verified by a ratio test. Figure 3.1b is an example of a circuit in $\mathbf{C}^{\mathsf{E}}$.

The transformation from $C \in \mathbf{C}^{\mathsf{S}}$ to $\tilde{C} \in \mathbf{C}^{\mathsf{E}}$ is as follows. The m inputs of C are multiplicatively shared among n parties to obtain $n \cdot m$ inputs for $\tilde{C}$; slot i of $\tilde{C}$ contains the m shares of party i. Each multiplication gate in C is mapped to $O(n)$ multiplication gates in $\tilde{C}$ tasked with assembling all the relevant shares; each addition gate in C is mapped to a corresponding addition gate in $\tilde{C}$. A crucial feature of the transformation is depth efficiency (see below).

The multi-party protocol for circuits in $\mathbf{C}^{\mathsf{E}}$ is a generalization of the one that we described above for a single party. Essentially, the class $\mathbf{C}^{\mathsf{E}}$ ensures that at each multiplication gate there is one party that knows the "local" witness for producing a NIZK proof of correct evaluation of the gate. The protocol proceeds in rounds, and at each round every party proves correct evaluation of any gate ready to be processed, and so on until no more gates need to be processed.

Finally, a subtle technical detail is that our multi-party protocol works for evaluating circuits in $\mathbf{C}^{\mathsf{E}}$ on most but not all inputs. Some inputs are pathological for our construction, so we rule them out; fortunately these arise with only negligible probability, so they do not bias the overall sampling protocol by more than a negligible amount. More precisely, we assume that the circuit wires will never carry zero values; this suffice for our applications and is easy to ensure in our construction.

Depth matters. The round complexity of securely evaluating $\tilde{C} \in \mathbf{C}^{\mathsf{E}}$ is $\mathsf{depth}_{\mathsf{E}}(\tilde{C}) + O(1)$, where $\mathsf{depth}_{\mathsf{E}}(\tilde{C})$ is the E-*depth* and (roughly) corresponds to the maximum number of *gate-ownership* alternations along any input-to-output path; ownership refers to which party provides the input share to a gate. (See Figure 3.1b for a comparison of depth and E-depth for an example in $\mathbf{C}^{\mathsf{E}}$.) Intuitively, while going down a path in the circuit, every change in gate ownership means that a party needs to wait on another one to process the previous gate, thereby costing an extra broadcast round.

Therefore, it is crucial that the transformation from C to $\tilde{C}$ is efficient in terms of E-depth of $\tilde{C}$. By carefully combining the subcircuits in $\tilde{C}$, we ensure that $\mathsf{depth}_{\mathsf{E}}(\tilde{C}) = n \cdot \mathsf{depth}_{\mathsf{S}}(C)$, where $\mathsf{depth}_{\mathsf{S}}(C)$ is the S-*depth* of C; S-*depth* denotes the maximum number of alternations between addition and multiplication gates along any input-to-output path. (See Figure 3.1a for a comparison of depth and S-depth for an example in $\mathbf{C}^{\mathsf{S}}$.)

3.2 Prior work

Prior work has not specifically studied the problem of parameter generation for zk-SNARKs, but has studied this several other cryptographic goals, as mentioned below. We also discuss other tools relevant to our security goals: secure multi-party computation and zero-knowledge proofs.

Addressing setup assumptions. Cryptographic constructions are sometimes proved secure contingent on the fact that certain *setup assumptions* hold (and *some* setup assumption is at times necessary). Prior work has studied the problem of relaxing setup assumptions for various cryptographic goals, including NIZKs and also, more generally, universally-composable (UC) security [Can01].

Canetti et al. [CPS07] study UC security in a model where the common reference string has been sampled by an adversary with some restrictions (e.g., the string is has enough min entropy and the sampling algorithm is efficient and known to the adversary). Groth and Ostrovsky [GO07] study NIZKs and UC security in a model where there are multiple common reference strings of which a majority are generated by honest parties. Goyal and Katz [GK08] study UC security in a model where there is a single common reference string with the twist that, if corrupted, security still holds provided not too many parties are also corrupted. Garg et al. [GGJS11] study UC security in a model where there are multiple setups at once and each party has a belief about which ones of these setups is trustworthy; this model captures as special cases the models of [GO07, GK08]. Katz et al. [KKZZ14] also study UC security in a related model where there are multiple setups (of which some fraction are corrupted).

Clark and Hengartner [CH10] study statistical properties of the stock market, in order to understand to what extent it can be used to obtain common random strings in practice (e.g., to be used for a NIZK).

Various works have also studied distributed generation of the discrete logarithm of a published value or of an RSA modulus [Ped92, CS04, GJKR07, KHG12, HMRT12]; such protocols are often useful in threshold cryptography.

Secure multi-party computation. The area of secure multi-party computation has seen rapid recent progress, both in terms of theoretical results and concrete implementations. Yet, the existing generic implementations do not support, or inefficiently support, the setting that we consider: many parties, dishonest majority (against non-adaptive corruptions), and evaluation of a circuit with large (standard) circuit depth.

For example, many implementations consider the case of two parties [MNPS04], where they achieve outstanding efficiency [sS13, BHKR13], and can process billions of boolean gates while spending only tens of CPU cycles on each. Most of the approaches in this setting are based on Yao's seminal work on garbled circuits [Yao86, LP09].

Some implementations consider the case of arbitrary number of parties, but they suffer from other limitations. For example, [BDNP08] consider adversaries that are honest but curious (but not malicious). Other protocols [BOGW88, DGKN09] consider malicious adversaries but require an honest majority. There are known constant-round MPC protocols for a fully-malicious, dishonest majority [KOS03, Pas04], but these require expensive ZK proofs and have not been implemented.

When requiring security against dishonest majorities (with at least one honest party), implementations have a round complexity that depends linearly on the depth of the circuit being computed [Orl11, DKL$^+$13, DLT14, BDOZ11, DPSZ12]. The applications that we consider in this paper involve circuit depths that exceed hundreds of thousands, resulting in large round complexities; such round complexities are at best very expensive when considering network latencies on the Internet and at worst prohibitive if one of the participating parties uses an air gap as a precaution. While theoretical results do achieve sublinear round complexity [AJL^{+}12, GGHR14], they rely on "heavy artillery" such as fully-homomorphic encryption and program obfuscation, unlikely to yield efficient implementations in the near future.

Finally, Baum et al. [BDO14] study the notion of auditable secure multi-party computation, in which anyone can use a verifier algorithm to ensure that the output is computed correctly even when all parties are corrupted. The multi-party protocol in this paper is auditable in this sense, but in our setting this property is not useful because we also care about the distribution of the output, for which we can say nothing when all parties are corrupted.

Zero-knowledge proofs. An idea to address the problem of generating public parameters is to task a single party to provide a zero-knowledge proof that the public parameters are an output of the generator algorithm. Yet, everyone else must still trust this party to (i) provide suitable randomness to the generator (so that the output is not only syntactically but also distributionally correct) (ii) not abuse or leak this randomness. Even if there were zero-knowledge proofs that are efficient for the NP statements that one would consider in our setting (e.g., [RMMY12, PRST08, MR14] could be good candidates), this use of zero-knowledge proofs does not address one of our main goals, which is distributing trust among multiple parties. That said, our construction can be intuitively viewed as a zero-knowledge proof that can be jointly computed by n parties of which only one must be honest; see Section 3.1.5.

3.3 Definitions

We give the definitions needed for technical discussions.

Remark 3.3.1. Some cryptographic primitives that we use (e.g., commitments and NIZKs) leverage only "simple" public parameters: pp is a random binary string of a certain length. In this case, pp is known as a *common random string*, and we denote it by crs. A common random string stands in contrast to more general forms of public parameters (which are a common *reference* string), leveraged by zk-SNARKs and whose generation is the problem that we set out to solve in the first place. It is thus not circular to primitives that require common *random* strings as part of our construction. (See Section 3.1.)

3.3.1 Basic notation

We denote by λ the security parameter; $f = O_\lambda(g)$ means that there exists $c > 0$ such that $f = O(\lambda^c g)$. The power set of a set S is denoted 2^S. Vectors are denoted by arrow-equipped letters (e.g., $\vec{a}$); their entries carry an index but not the arrow (e.g., a_1, a_2). Concatenation of vectors (and scalars) is denoted by the operator $\circ$.

Implicit inputs. To simplify notation, the input 1^λ is implicit to all cryptographic algorithms. Similarly, we do not make explicit adversaries' auxiliary inputs.

Distributions. We write $\{y \mid x_1 \leftarrow \mathcal{D}_1\,;\, x_2 \leftarrow \mathcal{D}_2\,;\, \dots\}_E$ to denote the distribution over y obtained by conditioning on the event E and sampling x_1 from $\mathcal{D}_1$, x_2 from $\mathcal{D}_2$, and so on, and then computing $y := y(x_1, x_2, \dots)$. Given two distributions $\mathcal{D}$ and $\mathcal{D}'$, we write $\mathcal{D} \overset{\mathsf{negl}}{=} \mathcal{D}'$ to denote that the statistical distance between $\mathcal{D}$ and $\mathcal{D}'$ is negligible in the security parameter λ. A distribution $\mathcal{D}$ is efficiently sampleable if there exists a probabilistic polynomial-time algorithm whose output follows the distribution $\mathcal{D}$.

Groups. We denote by $\mathbb{G}$ a group, and consider only groups that are cyclic and have a prime order r. Group elements are denoted with calligraphic letters (such as $\mathcal{P}, \mathcal{Q}$). We write $\mathbb{G} = \langle \mathcal{G} \rangle$ to denote that the element $\mathcal{G}$ generates $\mathbb{G}$, and use additive notation for group arithmetic. Hence, $\mathcal{P} + \mathcal{Q}$ denotes addition of the two elements $\mathcal{P}$ and $\mathcal{Q}$; $a \cdot \mathcal{P}$ denotes scalar multiplication of $\mathcal{P}$ by the scalar $a \in \mathbb{Z}$; and $\mathcal{O} := 0 \cdot \mathcal{P}$ denotes the identity element. Since $r \cdot \mathcal{P} = \mathcal{O}$, we can equivalently think of a scalar a as belonging to the field of size r. Given a vector of scalars $\vec{a} = (a_1, \dots, a_n)$, we use $\vec{a} \cdot \mathcal{P}$ as a shorthand for the vector $(a_1 \cdot \mathcal{P}, \dots, a_n \cdot \mathcal{P})$.

Fields. We denote by $\mathbb{F}$ a field, and by $\mathbb{F}_n$ the field of size n; we consider only fields of prime order. Field elements are denoted with Greek letters (such as α, β).

3.3.2 Commitments

A *commitment scheme* is a pair $\mathsf{COMM} = (\mathsf{COMM.G}, \mathsf{COMM.V})$ with the following syntax.

- $\underline{\mathsf{COMM.G}(\mathsf{crs}, x) \to (\mathsf{cm}, \mathsf{cr})}$: On input common random string crs and input data x, the *commitment generator* $\mathsf{COMM.G}$ probabilistically samples a commitment cm of x and corresponding commitment randomness cr.

- $\underline{\mathsf{COMM.V}(\mathsf{crs}, x, \mathsf{cm}, \mathsf{cr}) \rightarrow b}$: On input common random string crs, input data x, commitment cm, and commitment randomness cr, the *commitment verifier* COMM.V outputs $b = 1$ if cm is a commitment of x with randomness cr.

The common random string crs is COMM's public parameters, and consists of $O_\lambda(1)$ random bits.

The scheme COMM satisfies the standard completeness, (computational) binding, and (statistical) hiding properties. We do not assume that a commitment cm hides $|x|$ (i.e., we do not assume that COMM produces succinct commitments).

3.3.3 Non-interactive zero-knowledge proofs of knowledge

A *non-interactive zero-knowledge proof of knowledge* (NIZK) for an NP relation $\mathcal{R}$ in the common random string model is a tuple $\mathsf{NIZK}_{\mathcal{R}} = (\mathsf{NIZK}_{\mathcal{R}}.\mathsf{P}, \mathsf{NIZK}_{\mathcal{R}}.\mathsf{V}, \mathsf{NIZK}_{\mathcal{R}}.\mathsf{E}, \mathsf{NIZK}_{\mathcal{R}}.\mathsf{S})$ with the following syntax.

- $\underline{\mathsf{NIZK}_{\mathcal{R}}.\mathsf{P}(\mathsf{crs}, \mathbb{x}, \mathbb{w}) \rightarrow \pi}$: On input common random string crs, instance $\mathbb{x}$, and witness $\mathbb{w}$, the *prover* $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{P}$ outputs a non-interactive proof π for the statement "there is $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$".
- $\underline{\mathsf{NIZK}_{\mathcal{R}}.\mathsf{V}(\mathsf{crs}, \mathbb{x}, \pi) \rightarrow b}$: On input common random string crs, instance $\mathbb{x}$, and proof π, the *verifier* $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{V}$ outputs $b = 1$ if π is a convincing proof for the statement "there is $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$".

The common random string crs is $\mathsf{NIZK}_{\mathcal{R}}$'s public parameters, and consists of $O_\lambda(1)$ random bits. The remaining two components are each pairs of algorithms, as follows.

- $\underline{\mathsf{NIZK}_{\mathcal{R}}.\mathsf{E}_1 \rightarrow (\mathsf{crs}^{\mathsf{ext}}, \mathsf{trap}^{\mathsf{ext}})}$: The *extractor's generator* $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{E}_1$ samples a string $\mathsf{crs}^{\mathsf{ext}}$ (indistinguishable from crs) and corresponding trapdoor $\mathsf{trap}^{\mathsf{ext}}$.
- $\underline{\mathsf{NIZK}_{\mathcal{R}}.\mathsf{E}_2(\mathsf{crs}^{\mathsf{ext}}, \mathsf{trap}^{\mathsf{ext}}, \mathbb{x}, \pi) \rightarrow \mathbb{w}}$: On input $\mathsf{crs}^{\mathsf{ext}}$, $\mathsf{trap}^{\mathsf{ext}}$, instance $\mathbb{x}$, and proof π, the *extractor* $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{E}_2$ outputs a witness $\mathbb{w}$ for the instance $\mathbb{x}$.
- $\underline{\mathsf{NIZK}_{\mathcal{R}}.\mathsf{S}_1 \rightarrow (\mathsf{crs}^{\mathsf{sim}}, \mathsf{trap}^{\mathsf{sim}})}$: The *simulator's generator* $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{S}_1$ samples a string $\mathsf{crs}^{\mathsf{sim}}$ (indistinguishable from crs) and corresponding trapdoor $\mathsf{trap}^{\mathsf{sim}}$.
- $\underline{\mathsf{NIZK}_{\mathcal{R}}.\mathsf{S}_2(\mathsf{crs}^{\mathsf{sim}}, \mathsf{trap}^{\mathsf{sim}}, \mathbb{x}) \rightarrow \pi}$: On input $\mathsf{crs}^{\mathsf{sim}}$, $\mathsf{trap}^{\mathsf{sim}}$, and instance $\mathbb{x}$ (such that $\exists\, \mathbb{w} : (\mathbb{x}, \mathbb{w}) \in \mathcal{R}$), the *simulator* $\mathsf{NIZK}_{\mathcal{R}}.\mathsf{S}_2$ outputs a proof π (indistinguishable from an honest one).

The scheme $\mathsf{NIZK}_{\mathcal{R}}$ satisfies the standard completeness, (computational, adaptive) proof-of-knowledge, and (statistical, adaptive, multi-theorem) zero-knowledge properties. See, e.g., [FLS99, GOS06b, GOS06a] for formal definitions.

3.3.4 Arithmetic circuits

We consider *arithmetic*, rather than boolean, circuits. Given a field $\mathbb{F}$, an $\mathbb{F}$-*arithmetic circuit* C takes as input elements in $\mathbb{F}$, and its gates output elements in $\mathbb{F}$. We write $C\colon \mathbb{F}^m \to \mathbb{F}^h$ if C takes m inputs and produces h outputs.

Wires, inputs, gates, and size. We denote by $\mathsf{wires}(C)$ and $\mathsf{gates}(C)$ the wires and gates of C; also, we denote by $\mathsf{inputs}(C)$ and $\mathsf{outputs}(C)$ the subsets of $\mathsf{wires}(C)$ consisting of C's input and output wires. We denote by $\#\mathsf{wires}(C)$, $\#\mathsf{gates}(C)$,$\#\mathsf{inputs}(C)$, and $\#\mathsf{outputs}(C)$ the cardinalities of $\mathsf{wires}(C)$, $\mathsf{gates}(C)$, $\mathsf{inputs}(C)$, and $\mathsf{outputs}(C)$ respectively. The size of C is $\mathsf{size}(C) := \#\mathsf{inputs}(C) + \#\mathsf{gates}(C)$.

Gate types. A gate g of C is of one of three types:

- a *constant gate* that outputs a constant α with $\alpha \neq 0$;
- an *addition gate* that computes a linear combination $\sum_{j=1}^{d} \alpha_j \mathsf{w}_j$ with $d \geq 2$; or
- a *multiplication gate* that computes a product $\alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$ with $\alpha \neq 0$.

We define $\mathsf{type}(g)$ to be const for constant gates, add for addition gates, and mul for multiplication gates. If g is an addition gate, $\mathsf{inputs}(g) := \{\mathsf{w}_1, \ldots, \mathsf{w}_d\}$ are the input wires and $\mathsf{coeffs}(g) := (\alpha_1, \ldots, \alpha_d)$ are the coefficients. If g is a multiplication gate, $\mathsf{L\text{-}input}(g) := \mathsf{w}^{\mathsf{L}}$ is the left input wire, $\mathsf{R\text{-}input}(g) := \mathsf{w}^{\mathsf{R}}$ is the right input wire, and $\mathsf{coeffs}(g_{\mathsf{mul}}) := (\alpha)$ is the coefficient; also, $\mathsf{inputs}(g) := \{\mathsf{L\text{-}input}(g_{\mathsf{mul}}), \mathsf{R\text{-}input}(g_{\mathsf{mul}})\}$ are the two input wires. (By this use of sub/super-scripts, the r-th input of an addition gate can be easily distinguished from the right input of a multiplication gate.) For the three gate types, $\mathsf{output}(g) := \mathsf{w}$ is the output wire; also, g_{w} is the gate for which $\mathsf{w} = \mathsf{output}(g_{\mathsf{w}})$. As usual, the dependency graph induced by C's gates is acyclic. Moreover, each gate g of circuit C computes a polynomial in $\mathsf{inputs}(C)$; we use $\mathsf{wire\text{-}poly}(\mathsf{w})$ to denote the polynomial computed by g_{w}. We also use $\#\mathsf{const\text{-}gates}(C)$, $\#\mathsf{add\text{-}gates}(C)$, and $\#\mathsf{mul\text{-}gates}(C)$ to denote the number of constant, addition, and multiplication gates (and, of course, it then holds that $\#\mathsf{gates}(C) = \#\mathsf{const\text{-}gates}(C) + \#\mathsf{add\text{-}gates}(C) + \#\mathsf{mul\text{-}gates}(C)$).

Valid inputs. Given a circuit $C\colon \mathbb{F}^m \to \mathbb{F}^h$, at times we consider only a subset of all of its possible inputs in $\mathbb{F}^m$; these are the inputs that we consider to be "valid". We think of this set as part of the description of C and denote it by $\mathsf{valid\text{-}inputs}(C)$.

Further notions for circuits with partitioned domains. We also consider $\mathbb{F}$-arithmetic circuits C for which the m inputs of the circuits are partitioned into n disjoint *slots*; in such a case, we write $C\colon \mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n} \to \mathbb{F}^h$ to express that the first m_1 inputs belong to the first slot, the next m_2 to the second, and so on; the integers $m_1, \ldots, m_n$ are then also part of

C's description. In our protocols, the i-th slot will carry the inputs of the i-th participant of the protocol. For $i = 1, \ldots, n$: we denote by $\mathsf{inputs}(C,i)$ the input wires that belong to the i-th slot, and by $\mathsf{gates}(C,i)$ the gates that take as input an input wire in $\mathsf{inputs}(C,i)$; the notations $\#\mathsf{inputs}(C,i)$ and $\#\mathsf{gates}(C,i)$ denote the cardinalities of these sets; and we define $\mathsf{size}(C,i) := \#\mathsf{inputs}(C,i) + \#\mathsf{gates}(C,i)$. For every $\mathsf{w} \in \mathsf{inputs}(C)$, $\mathsf{slot}(C,\mathsf{w})$ is w's slot number, i.e., the index i such that $\mathsf{w} \in \mathsf{inputs}(C,i)$.

Finally, to assist in stating the definition of E-depth (see below), we introduce the *dependency set* $\mathsf{ds}(\mathsf{w})$ of a wire w; roughly, it denotes the subset of $\{1, \ldots, n\}$ comprising the slots that individually carry enough information (in terms of inputs) to compute the value of w in the most efficient evaluation schedule.

The formal definition of $\mathsf{ds}(\mathsf{w})$ is quite technical, and is given below:

$$\mathsf{ds}(\mathsf{w}) :=$$

$$\begin{cases} \{\mathsf{slot}(C,\mathsf{w})\} & \text{if } \mathsf{w} \in \mathsf{inputs}(C) \\ \{1, \ldots, n\} & \text{if } \mathsf{w} \notin \mathsf{inputs}(C), \mathsf{type}(g_{\mathsf{w}}) = \mathsf{const} \\ \{1, \ldots, n\} & \text{if } \mathsf{w} \notin \mathsf{inputs}(C), \mathsf{type}(g_{\mathsf{w}}) = \mathsf{add}, \bigcap_{\mathsf{w}' \in \mathsf{inputs}(g_{\mathsf{w}})} \mathsf{ds}(\mathsf{w}') = \emptyset \\ \bigcap_{\mathsf{w}' \in \mathsf{inputs}(g_{\mathsf{w}})} \mathsf{ds}(\mathsf{w}') & \text{if } \mathsf{w} \notin \mathsf{inputs}(C), \mathsf{type}(g_{\mathsf{w}}) = \mathsf{add}, \bigcap_{\mathsf{w}' \in \mathsf{inputs}(g_{\mathsf{w}})} \mathsf{ds}(\mathsf{w}') \neq \emptyset \\ \mathsf{ds}(\mathsf{L\text{-}input}(g_{\mathsf{w}})) & \text{if } \mathsf{w} \notin \mathsf{inputs}(C), \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ does not depend on any inputs} \\ \{i\} & \text{if } \mathsf{w} \notin \mathsf{inputs}(C), \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ only depends on inputs from slot } i \end{cases}$$

Two classes of circuits. Below we define the two circuit classes $\mathbf{C}^{\mathsf{S}}$ and $\mathbf{C}^{\mathsf{E}}$.

- $\mathbf{C}^{\mathsf{S}}$ is the class of $\mathbb{F}$-arithmetic circuits $C \colon \mathbb{F}^m \to \mathbb{F}^h$ for which every gate g in $\mathsf{gates}(C)$ is such that:
  (i) $\mathsf{output}(g) \in \mathsf{outputs}(C)$;
  (ii) if $\mathsf{type}(g) = \mathsf{add}$, then $\mathsf{inputs}(g) \cap \mathsf{inputs}(C) = \emptyset$; and
  (iii) if $\mathsf{type}(g) = \mathsf{mul}$, then $\mathsf{L\text{-}input}(g) \notin \mathsf{inputs}(C)$ and either $\mathsf{R\text{-}input}(g) \in \mathsf{inputs}(C)$ or $\mathsf{R\text{-}input}(g)$ is the output of an addition-free subcircuit.
- $\mathbf{C}^{\mathsf{E}}$ is the class of $\mathbb{F}$-arithmetic circuits $C \colon \mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n} \to \mathbb{F}^h$ for which every gate g in $\mathsf{gates}(C)$ is such that:
  (i) if $\mathsf{type}(g) = \mathsf{add}$, then $\mathsf{inputs}(g) \subseteq \mathsf{outputs}(C)$ and $\mathsf{output}(g) \in \mathsf{outputs}(C)$;
  (ii) if $\mathsf{type}(g) = \mathsf{mul}$, then $\mathsf{L\text{-}input}(g) \notin \mathsf{inputs}(C)$ and for every $\mathsf{w}, \mathsf{w}' \in \mathsf{inputs}(C)$, if $\mathsf{R\text{-}input}(g)$ depends on w and w' then $\mathsf{slot}(C,\mathsf{w}) = \mathsf{slot}(C,\mathsf{w}')$; and
  (iii) if $\mathsf{type}(g) = \mathsf{mul}$ and $\mathsf{output}(g) \in \mathsf{outputs}(C)$, then $\mathsf{L\text{-}input}(g) \in \mathsf{outputs}(C)$ and either $\mathsf{R\text{-}input}(g) \in \mathsf{inputs}(C)$ or there exist three wires $\mathsf{w}_1, \mathsf{w}_2, \mathsf{w}_3 \in \mathsf{outputs}(C)$ such that $\mathsf{wire\text{-}poly}(\mathsf{output}(g)) = \mathsf{wire\text{-}poly}(\mathsf{w}_1) \cdot \frac{\mathsf{wire\text{-}poly}(\mathsf{w}_2)}{\mathsf{wire\text{-}poly}(\mathsf{w}_3)}$. Moreover, there exists a slot i such that all input dependencies of $\mathsf{R\text{-}input}(g)$, w_2 and w_3 belong to the slot

i.

The last case captures non-output gates, where the correct computation can be checked via a ratio test for two output wires w_2 and w_3. We introduce a special notation for referencing the existentially quantified w_1, w_2 and w_3, letting $\mathsf{mul\text{-}wit}_1(C,g)$, $\mathsf{mul\text{-}wit}_2(C,g)$ and $\mathsf{mul\text{-}wit}_3(C,g)$ denote a (canonical) choice of w_1, w_2 and w_3, respectively; these are part of the circuit description. When constructing a circuit C in $\mathbf{C}^{\mathsf{E}}$ we will specify a concrete choice for $\mathsf{mul\text{-}wit}_1(C,g)$, $\mathsf{mul\text{-}wit}_2(C,g)$ and $\mathsf{mul\text{-}wit}_3(C,g)$, to prove its existence, but we stress that our protocols work for *any* choice of w_1, w_2 and w_3 that satisfies the restrictions above.

In the third case above, a wire $\mathsf{mul\text{-}wit}_3(C,g)$ appears as a denominator, so we need to exclude inputs to the circuit that cause it to be zero. For circuits C in the class $\mathbf{C}^{\mathsf{E}}$ we set $\mathsf{valid\text{-}inputs}(C)$ to be the subset of $\mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n}$ such that for every multiplication-by-output gate g and every $\vec{\alpha} \in \mathsf{valid\text{-}inputs}(C)$ we have $\mathsf{wire\text{-}poly}(\mathsf{mul\text{-}wit}_3(C,g))(\vec{\alpha}) \neq 0$.

Finally, we stipulate that for every circuit C in the class $\mathbf{C}^{\mathsf{E}}$, the directed graph formed by its gates (as vertices), their output wires (as edges) and additional edges connecting $\mathsf{mul\text{-}wit}_1(C,g)$, $\mathsf{mul\text{-}wit}_2(C,g)$, and $\mathsf{mul\text{-}wit}_3(C,g)$ to g, must be acyclic.

Notions of depth. For circuits in $\mathbf{C}^{\mathsf{S}}$ and $\mathbf{C}^{\mathsf{E}}$, we use alternative notions of depth, called S-*depth* and E-*depth*; both S-depth and E-depth are bounded from above by (traditional) circuit depth, but are sometimes much less than it. (For example, the circuit computing $\mathsf{w} \to (1, \mathsf{w}, \mathsf{w}^2, \ldots, \mathsf{w}^{100})$ has depth 100, while it has S-depth and E-depth equal to 1.)

- The S-depth of C in $\mathbf{C}^{\mathsf{S}}$ is $\mathsf{depth}_{\mathsf{S}}(C) := \max_{\mathsf{w} \in \mathsf{outputs}(C)} \mathsf{depth}_{\mathsf{S}}(\mathsf{w})$, where $\mathsf{depth}_{\mathsf{S}}(\mathsf{w})$ measures the greatest number of alternations along a path from inputs between (a) multiplication gates for which the right input wire depends on an input wire; and (b) all other gates. A formal definition follows.

  When $\mathsf{w} \in \mathsf{inputs}(C)$, we define $\mathsf{depth}_{\mathsf{S}}(\mathsf{w}) := 0$, and when $\mathsf{w} \notin \mathsf{inputs}(C)$ we set

$$\mathsf{depth}_{\mathsf{S}}(\mathsf{w}) := \begin{cases} 0 & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{const} \\ \max\ \{\mathsf{depth}_{\mathsf{S}}(\mathsf{w}')\}_{\mathsf{w}' \in \mathsf{inputs}(g_{\mathsf{w}})} & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{add} \\ \mathsf{depth}_{\mathsf{S}}(\mathsf{L\text{-}input}(g_{\mathsf{w}})) & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ does not depend on any inputs} \\ \mathsf{depth}_{\mathsf{S}}(\mathsf{L\text{-}input}(g_{\mathsf{w}})) + b^{\mathsf{S}}_{\mathsf{L\text{-}input}(g_{\mathsf{w}})} & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ depends on at least one input} \end{cases} .$$

  Above $b^{\mathsf{S}}_{\mathsf{w}} \in \{0,1\}$ is set to be 1 for $\mathsf{w} \in \mathsf{inputs}(C)$, and is defined recursively for $\mathsf{w} \notin \mathsf{inputs}(C)$:

$$b^{\mathsf{S}}_{\mathsf{w}} := \begin{cases} 1 & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{const} \\ 1 & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{add} \\ b^{\mathsf{S}}_{\mathsf{L\text{-}input}(g_{\mathsf{w}})} & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ does not depend on any inputs} \\ 0 & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ depends on at least one input} \end{cases}.$$

Remark 3.3.2. When $\mathsf{w} \in \mathsf{inputs}(C)$, the values of $\mathsf{depth}_{\mathsf{S}}(\mathsf{w})$ and $b^{\mathsf{S}}_{\mathsf{w}}$ can be defined arbitrarily, as they are not referenced in any calculations. For symmetry with $\mathsf{depth}_{\mathsf{E}}(\mathsf{w})$ we define both values to equal 1.

- The E-depth of C in $\mathbf{C}^{\mathsf{E}}$ is $\mathsf{depth}_{\mathsf{E}}(C) := \max_{\mathsf{w} \in \mathsf{outputs}(C)} \mathsf{depth}_{\mathsf{E}}(\mathsf{w})$. The depth $\mathsf{depth}_{\mathsf{E}}(\mathsf{w})$ captures the minimum number of broadcast rounds required to obtain an encoded evaluation of w, and counts the number of gates on a path to w with conflicting temporal dependencies (i.e., with dependency sets whose intersection is empty).
  Again, a formal technical definition follows.
  We set $\mathsf{depth}_{\mathsf{E}}(\mathsf{w}) := 0$ for $\mathsf{w} \in \mathsf{inputs}(C)$ and for all other $\mathsf{w} \notin \mathsf{inputs}(C)$ define $\mathsf{depth}_{\mathsf{E}}(\mathsf{w})$ as follows:

$$\mathsf{depth}_{\mathsf{E}}(\mathsf{w}) :=$$
$$\begin{cases} 0 & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{const} \\ \max_{\mathsf{w}' \in \mathsf{inputs}(g_{\mathsf{w}})} \mathsf{depth}_{\mathsf{E}}(\mathsf{w}') + b^{\mathsf{add}}_{\mathsf{w}} & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{add} \\ \mathsf{depth}_{\mathsf{E}}(\mathsf{L\text{-}input}(g_{\mathsf{w}})) & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ does not depend on any inputs} \\ \mathsf{depth}_{\mathsf{E}}(\mathsf{L\text{-}input}(g_{\mathsf{w}})) + b^{\mathsf{inp}}_{\mathsf{w}} & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ is a circuit input} \\ \mathsf{depth}_{\mathsf{E}}(\mathsf{mul\text{-}wit}_1(C, g_{\mathsf{w}})) + b^{\mathsf{priv}}_{\mathsf{w}} & \text{if } \mathsf{type}(g_{\mathsf{w}}) = \mathsf{mul} \text{ and } \mathsf{R\text{-}input}(g_{\mathsf{w}}) \text{ is a non-constant, non-input wire of } C \end{cases}$$

  where:
  - $b^{\mathsf{add}}_{\mathsf{w}} \in \{0,1\}$ is 1 if and only if $\bigcap_{\mathsf{w}' \in \mathsf{inputs}(g_{\mathsf{w}})} \mathsf{ds}(\mathsf{w}') = \emptyset$; and
  - $b^{\mathsf{inp}}_{\mathsf{w}} \in \{0,1\}$ is 1 if and only if $\mathsf{ds}(\mathsf{L\text{-}input}(g_{\mathsf{w}})) \cap \mathsf{ds}(\mathsf{R\text{-}input}(g_{\mathsf{w}})) = \emptyset$ or $\mathsf{L\text{-}input}(g_{\mathsf{w}})$ does not depend on any inputs.
  - $b^{\mathsf{priv}}_{\mathsf{w}} \in \{0,1\}$ is 1 if and only if $\mathsf{ds}(\mathsf{mul\text{-}wit}_1(C, g_{\mathsf{w}})) \cap \mathsf{ds}(\mathsf{R\text{-}input}(g_{\mathsf{w}})) = \emptyset$ or $\mathsf{L\text{-}input}(g_{\mathsf{w}})$ does not depend on any inputs. Note that in this case we have $\mathsf{ds}(\mathsf{R\text{-}input}(g_{\mathsf{w}})) = \mathsf{ds}(\mathsf{mul\text{-}wit}_2(C, g_{\mathsf{w}})) = \mathsf{ds}(\mathsf{mul\text{-}wit}_3(C, g_{\mathsf{w}})) = \{i\}$, where i is the party on whose inputs $\mathsf{R\text{-}input}(g_{\mathsf{w}})$ depends on.

3.3.5 Pairings and duplex-pairing groups

Pairings. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be cyclic groups of a prime order r. Let $\mathcal{G}_1$ be a generator of $\mathbb{G}_1$ (i.e., $\mathbb{G}_1 = \{\alpha \mathcal{G}_1\}_{\alpha \in \mathbb{F}_r}$) and let $\mathcal{G}_2$ be a generator for $\mathbb{G}_2$. A *pairing* is an efficient map

$e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_T$ is also a cyclic group of order r (which, unlike other groups, we write in multiplicative notation), satisfying the following properties.

- BILINEARITY. For every pair of nonzero elements $\alpha, \beta \in \mathbb{F}_r$, it holds that $e(\alpha\mathcal{G}_1, \beta\mathcal{G}_2) = e(\mathcal{G}_1, \mathcal{G}_2)^{\alpha\beta}$.
- NON-DEGENERACY. $e(\mathcal{G}_1, \mathcal{G}_2)$ is not the identity in $\mathbb{G}_T$.

Duplex-pairing groups. A group $\mathbb{G}$ of prime order r is *duplex pairing* if there are order-r groups $\mathbb{G}_1$ and $\mathbb{G}_2$ such that (i) there is a pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for some target group $\mathbb{G}_T$, and (ii) there are generators $\mathcal{G}_1$ of $\mathbb{G}_1$ and $\mathcal{G}_2$ of $\mathbb{G}_2$ such that $\mathbb{G}$ is isomorphic to $\{(\alpha\mathcal{G}_1, \alpha\mathcal{G}_2) \mid \alpha \in \mathbb{F}_r\} \subseteq \mathbb{G}_1 \times \mathbb{G}_2$.

3.4 Secure multi-party computation

The security goal of our sampling protocol is formalized, via the language of secure multi-party computation, as realizing a certain ideal functionality. We specialize standard definitions of secure multi-party computation [GMW87b, BOGW88] to our setting, by considering parties' inputs that are field elements rather than bit strings, by considering families of functionalities rather than a single functionality, and making explicit the notion of a (transcript) verifier. These definitions provide background and notation for this paper (and closely follow the treatment in [AL11]). We assume familiarity with simulation-based security definitions; for more, see [AL11].

3.4.1 Multi-party broadcast protocols with common random strings

We consider multi-party protocols that run over a synchronous network with an authenticated broadcast channel and a common random string. Before the protocol begins, a random string of a certain prescribed length, denoted crs, is made available to all parties; to simplify notation, we do not make crs an explicit input. Afterwards, the protocol proceeds in rounds and, at each round, the protocol's schedule determines which parties act; a party acts by broadcasting a message to all other parties. The broadcast channel is authenticated in that all parties always know who sent a particular message (regardless of what an adversary may do). We now introduce some notations and notions for later discussions.

Honest execution. Given a positive integer n, an n-**party broadcast protocol** is a tuple $\Pi = (S, \Sigma_1, \ldots, \Sigma_n)$ where $S\colon \mathbb{N} \to 2^{\{1,\ldots,n\}}$ is the deterministic polynomial-time *schedule* function and, for $i = 1, \ldots, n$, Σ_i is the (possibly stateful) probabilistic polynomial-time *strategy* of party i.

The *execution* of Π on an input $\vec{x} = (x_1, \ldots, x_n)$, denoted $[\![\Pi, \vec{x}]\!]$, works as follows. Set $t := 1$. While $S(t) \neq \emptyset$: (i) for each $i \in S(t)$ in any order, party i runs Σ_i, on input (x_i, t) and with oracle access to the history of messages broadcast so far, and broadcasts the resulting output message $\mathsf{msg}_{t,i}$ and, then, (ii) t increases by 1.

The *transcript* of $[\![\Pi, \vec{x}]\!]$, denoted tr, is the sequence of triples $(t, i, \mathsf{msg}_{t,i})$ ordered by $\mathsf{msg}_{t,i}$'s broadcast time. The *output* of $[\![\Pi, \vec{x}]\!]$, denoted out, is the last message in the transcript. Since Π's strategies are probabilistic, the transcript and output of $[\![\Pi, \vec{x}]\!]$ are random variables.

The *round complexity* is $\mathsf{ROUND}(\Pi) := \min_{t \in \mathbb{N}}\{t \mid S(t+1) = \emptyset\}$. For $i = 1, \ldots, n$, the *time complexity of party* i is $\mathsf{TIME}(\Pi, i) := \sum_{t \in [\mathsf{ROUND}(\Pi)] \text{ s.t. } i \in S(t)} \mathsf{TIME}(\Sigma_i, t)$ where $\mathsf{TIME}(\Sigma_i, t)$ is $\Sigma_i(\cdot, t)'$ time complexity. In our construction, the worst-case time complexity matches the expected time complexity.

Adversarial execution. Let A be a probabilistic polynomial-time algorithm and J a subset of $\{1, \ldots, n\}$. We denote by $[\![\Pi, \vec{x}]\!]_{A,J}$ the execution $[\![\Pi, \vec{x}]\!]$ modified so that A controls parties in J, i.e., A knows the private states of parties in J, may alter the strategies of parties in J, and may wait, in each round, to first see the messages broadcast by parties not in J and, only after that, instruct parties in J to send their messages. (In particular, $[\![\Pi, \vec{x}]\!]_{A,\emptyset} = [\![\Pi, \vec{x}]\!]$.) We denote by $\mathsf{REAL}_{\Pi,A,J}(\vec{x})$ the concatenation of the output of $[\![\Pi, \vec{x}]\!]_{A,J}$ and the view of A in $[\![\Pi, \vec{x}]\!]_{A,J}$.

3.4.2 Ideal functionalities

While Section 3.4.1 describes the real-world execution of a protocol Π on an input $\vec{x}$, here we describe the ideal-world execution of a function f on an input $\vec{x}$: each party i privately sends his input x_i to a trusted party, who broadcasts $f(\vec{x})$.

Adversarial execution. Let S be a probabilistic polynomial-time algorithm and J a subset of $\{1, \ldots, n\}$. The ideal-world execution of f on $\vec{x}$ when S controls parties in J differs from the above one as follows: S may substitute the inputs of parties in J with other same-length inputs. We denote by $\mathsf{IDEAL}_{f,S,J}(\vec{x})$ the concatenation of the value broadcast by the trusted party and the output of S in the ideal-world execution of f on $\vec{x}$ when S controls parties in J.

3.4.3 Secure sampling broadcast protocols

We define the notion of a secure sampling broadcast protocol, which captures our main security goal, i.e., a multi-party protocol for securely sampling the encoding of the random evaluation of a given circuit.

Let r be a prime, $\mathbb{G} = \langle \mathcal{G} \rangle$ an order-r group, n a positive integer, and $C \colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ an $\mathbb{F}_r$-arithmetic circuit. A **secure sampling broadcast protocol with n parties for C over $\mathbb{G}$** is a tuple $\Pi^{\mathsf{S}} = (\Pi, V, S)$, where Π is an n-party broadcast protocol, and V (the *verifier*) and S (the *simulator*) are probabilistic polynomial-time algorithms, that satisfies the following.

For every probabilistic polynomial-time algorithm A (the *adversary*) and subset J of $\{1, \ldots, n\}$ (the *corrupted parties*) with $|J| < n$ (i.e., with at least one honest party), it holds that

$$\left\{ \mathsf{REAL}_{\Pi,A,J}(\vec{\sigma}) \;\middle|\; \begin{matrix} \vec{\sigma}_1 \leftarrow \mathbb{F}_r^m \\ \vdots \\ \vec{\sigma}_n \leftarrow \mathbb{F}_r^m \end{matrix} \right\}_{V=1} \overset{\mathsf{negl}}{=} \left\{ \mathsf{IDEAL}_{f^{\mathsf{S}}_{n,C,\mathcal{G}},S(A,J),J}(\vec{\sigma}) \;\middle|\; \begin{matrix} \vec{\sigma}_1 \leftarrow (\mathbb{F}_r^*)^m \\ \vdots \\ \vec{\sigma}_n \leftarrow (\mathbb{F}_r^*)^m \end{matrix} \right\}_{S \neq \mathsf{abort}} ,$$

where:

- $\vec{\sigma}$ denotes $(\vec{\sigma}_1, \ldots, \vec{\sigma}_n)$;
- $V = 1$ denotes conditioning on the event that V, on input the transcript of $[\![\Pi, \vec{x}]\!]_{A,J}$, outputs 1;
- $S \neq \mathsf{abort}$ denotes the event that S in the ideal-world execution of $f^{\mathsf{S}}_{n,C,\mathcal{G}}$ on $\vec{\sigma}$ does not output the special symbol abort; and
- $f^{\mathsf{S}}_{n,C,\mathcal{G}}$ denotes the deterministic function such that $f^{\mathsf{S}}_{n,C,\mathcal{G}}(\sigma) := C((\prod_{i=1}^{n} \sigma_{i,1}, \ldots, \prod_{i=1}^{n} \sigma_{i,m})) \cdot \mathcal{G}$.

Extending the definition to variable number of parties and restricted circuit classes. Let r be a prime, $\mathbb{G} = \langle \mathcal{G} \rangle$ a group of order r, and $\mathbf{C}$ a class of $\mathbb{F}_r$-arithmetic circuits. A **secure sampling broadcast protocol for $\mathbf{C}$ over $\mathbb{G}$** is a tuple $\Pi^{\mathsf{S}} = (\Pi, V, S)$ such that, for every positive integer n and circuit $C \colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ in $\mathbf{C}$, $(\Pi_{n,C}, V_{n,C}, S_{n,C})$ is a secure sampling broadcast protocol with n parties for C over $\mathbb{G}$.

3.4.4 Secure evaluation broadcast protocols

We define the notion of a secure evaluation broadcast protocol, which captures an intermediate security goal, consisting of a multi-party protocol for jointly computing the encoding of the evaluation of a given circuit whose inputs are split into slots, with one slot per party.

(As described in Section 3.5, we reduce sampling to evaluation.)

Let r be a prime, $\mathbb{G} = \langle \mathcal{G} \rangle$ an order-r group, n a positive integer, and $C\colon \mathbb{F}_r^{m_1} \times \cdots \times \mathbb{F}_r^{m_n} \to \mathbb{F}_r^h$ an $\mathbb{F}_r$-arithmetic circuit. A **secure evaluation broadcast protocol with n parties for C over $\mathbb{G}$** is a tuple $\Pi^{\mathsf{E}} = (\Pi, V, S)$, where Π is an n-party broadcast protocol, and V (the *verifier*) and S (the *simulator*) are probabilistic polynomial-time algorithms, that satisfies the following.

For every probabilistic polynomial-time algorithm A (the *adversary*), subset J of $\{1, \ldots, n\}$ (the *corrupted parties*) with $|J| < n$ (i.e., with at least one honest party), and input $\vec{\sigma} = (\vec{\sigma}_1, \ldots, \vec{\sigma}_n)$ in $\mathbb{F}_r^{m_1} \times \cdots \times \mathbb{F}_r^{m_n}$, it holds that

$$\left\{\mathsf{REAL}_{\Pi,A,J}(\vec{\sigma})\right\}_{V=1} \overset{\mathsf{negl}}{=} \left\{\mathsf{IDEAL}_{f^{\mathsf{E}}_{C,\mathcal{G}},S(A,J),J}(\vec{\sigma})\right\}_{S \neq \mathsf{abort}} .$$

where:

- $V = 1$ denotes the event that V, on input the transcript of $[\![\Pi, \vec{x}]\!]_{A,J}$, outputs 1;
- $S \neq \mathsf{abort}$ denotes the event that S in the ideal-world execution of $f^{\mathsf{E}}_{C,\mathcal{G}}$ on $\vec{\sigma}$ does not output the special symbol abort; and
- $f^{\mathsf{E}}_{C,\mathcal{G}}$ denotes the deterministic function such that $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma})$ takes the value $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}) := \bot$ if $\vec{\sigma} \notin \mathsf{valid\text{-}inputs}(C)$, and otherwise takes the value $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}) := C(\vec{\sigma}) \cdot \mathcal{G}$.

Extending the definition to variable number of parties and restricted circuit classes. Let r be a prime, $\mathbb{G} = \langle \mathcal{G} \rangle$ a group of order r, and $\mathbf{C}$ a class of $\mathbb{F}_r$-arithmetic circuits. A **secure evaluation broadcast protocol for $\mathbf{C}$ over $\mathbb{G}$** is a tuple $\Pi^{\mathsf{E}} = (\Pi, V, S)$ such that, for every positive integer n and circuit $C\colon \mathbb{F}_r^{m_1} \times \cdots \times \mathbb{F}_r^{m_n} \to \mathbb{F}_r^h$ in $\mathbf{C}$, $(\Pi_{n,C}, V_{n,C}, S_{n,C})$ is a secure evaluation broadcast protocol with n parties for C over $\mathbb{G}$.

3.5 Secure sampling for a class of circuits

Our main construction is a multi-party protocol for securely sampling values of the form $C(\vec{\alpha}) \cdot \mathcal{G}$ for a random $\vec{\alpha}$, provided that C belongs to the class $\mathbf{C}^{\mathsf{S}}$. We use two cryptographic ingredients: commitment schemes (see Section 3.3.2) and NIZKs (see Section 3.3.3); both rely on a common *random* string, available in our setting (see Section 3.4.1).

Theorem 3.5.1. *Assume the existence of commitment schemes and NIZKs. Let r be a prime and $\mathbb{G}$ a group of order r. There is a secure sampling broadcast protocol $\Pi^{\mathsf{S}} = (\Pi, V, S)$ for $\mathbf{C}^{\mathsf{S}}$ over $\mathbb{G}$ such that, for every positive integer n and circuit C in $\mathbf{C}^{\mathsf{S}}$, the following holds.*

- *Round efficiency:* $\mathsf{ROUND}(\Pi_{n,C}) = n \cdot \mathsf{depth}_{\mathsf{S}}(C) + 3$.

- *Execution efficiency: for $i = 1, \ldots, n$, $\mathsf{TIME}(\Pi_{n,C}, i) = O_\lambda(\mathsf{size}(C))$.*
- *Verification efficiency: $V_{n,C}$ runs in time $O_\lambda(n \cdot \mathsf{size}(C))$.*
- *Simulation efficiency: $S_{n,C}$ runs in time $O_\lambda(n \cdot \mathsf{size}(C))$.*

Later on we discuss efficient instantiations for the above commitment schemes and NIZKs (when further relying on random oracles); see Section 3.6. Also, our implementation and evaluation target the case when $\mathbb{G}$ is a duplex-pairing group (defined in Section 3.3.5); see Section 3.7 and Section 3.8.

Proof strategy. We construct the protocol of Theorem 3.5.1 in two steps. The first step (Lemma 3.5.2) is a reduction from the problem of constructing secure *sampling* broadcast protocols to the problem of constructing secure *evaluation* broadcast protocols. The second step (Lemma 3.5.3) is a construction of such a secure evaluation broadcast protocol.

Lemma 3.5.2 (Sampling-to-evaluation reduction). *Let r be a prime and $\mathbb{G}$ a group of order r. There exist polynomial-time transformations T_1 and T_2 for which the following holds.*

- *For every positive integer n and circuit C in $\mathbf{C}^{\mathsf{S}}$:*
  - *$\tilde{C} := T_1(n, C)$ is a circuit in $\mathbf{C}^{\mathsf{E}}$;*
  - *$\Pi^{\mathsf{S}} := T_2(\Pi^{\mathsf{E}})$ is a secure sampling broadcast protocol with n parties for C over $\mathbb{G}$ for every Π^{E} that is a secure evaluation broadcast protocol with n parties for C over $\mathbb{G}$.*
- *T_1 builds a new circuit $\tilde{C}$ that is not much "costlier" than C:*
  - *$\mathsf{depth}_{\mathsf{E}}(\tilde{C}) = n \cdot \mathsf{depth}_{\mathsf{S}}(C)$;*
  - *$\mathsf{size}(\tilde{C}) = O(n \cdot \mathsf{size}(C))$; and*
  - *$\mathsf{size}(\tilde{C}, i) = O(\mathsf{size}(C))$ for $i = 1, \ldots, n$.*
- *T_2 increases the protocol's round complexity by 1, and preserves all time complexities up to $O_\lambda(1)$ factors.*

Lemma 3.5.3 (Evaluation protocol). *Assume the existence of commitment schemes and NIZKs. Let r be a prime and $\mathbb{G}$ a group of order r. There is a secure evaluation broadcast protocol $\Pi^{\mathsf{E}} = (\Pi, V, S)$ for $\mathbf{C}^{\mathsf{E}}$ over $\mathbb{G}$ such that, for every positive integer n and circuit C in $\mathbf{C}^{\mathsf{E}}$:*

- *$\mathsf{ROUND}(\Pi_{n,C}) = \mathsf{depth}_{\mathsf{E}}(C) + 2$;*
- *$\mathsf{TIME}(\Pi_{n,C}, i) = O_\lambda(\mathsf{size}(C, i))$ for $i = 1, \ldots, n$; and*
- *$V_{n,C}$ and $S_{n,C}$ run in time $O_\lambda(\mathsf{size}(C))$.*

Proofs of Lemma 3.5.2 and Lemma 3.5.3 are given in Appendix 3.10 and Appendix 3.11, and sketched below.

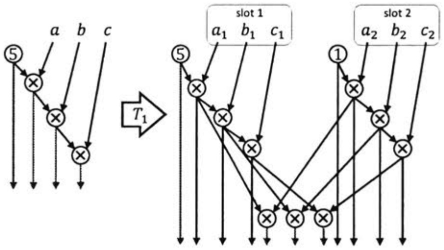
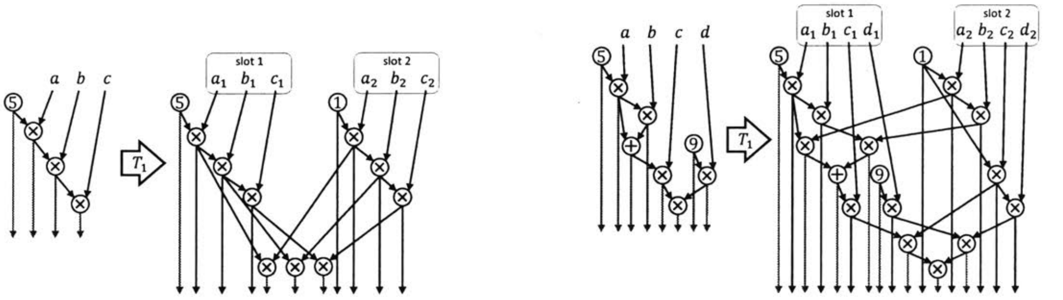


Figure 3.2: Example of a circuit C in $\mathbf{C}^{\mathsf{S}}$ and the corresponding circuit $\tilde{C} := T_1(C, n)$ in $\mathbf{C}^{\mathsf{E}}$ for $n = 2$ parties. This toy example corresponds to an "failed attempt" (see Section 3.5.1) as the straight-forward duplicate and multiply strategy does not preserve zero-knowledge when generalized.



(a) Example where C has only multiplication gates.

(b) Example where C has both addition and multiplication gates.

Figure 3.3: Two examples of a circuit C in $\mathbf{C}^{\mathsf{S}}$ and the corresponding circuit $\tilde{C} := T_1(C, n)$ in $\mathbf{C}^{\mathsf{E}}$ for $n = 2$ parties. The blue arrows in C denote the output wires of C; the blue arrows in $\tilde{C}$ denote the output wires of $\tilde{C}$ that compute outputs of C (while the remaining output wires carry partial computations).

3.5.1 Sketch of the sampling-to-evaluation reduction

We sketch the proof of Lemma 3.5.2. At a high level, the two transformations T_1 and T_2 work as follows.

- The circuit transformation T_1, given the number of parties n and a circuit C in $\mathbf{C}^{\mathsf{S}}$, outputs a circuit $\tilde{C} \in \mathbf{C}^{\mathsf{E}}$ that computes C's output, along with other auxiliary values, by suitably combining n multiplicative shares of C's input. (The alternative option of additive sharing results in worse efficiency parameters for $\tilde{C}$.)
- The protocol transformation T_2, given a secure evaluation protocol Π^{E} for $\tilde{C}$, outputs a secure sampling protocol Π^{S} for C by: (i) generating random shares for all inputs, to ensure uniform sampling; (ii) extending the protocol by one last round, to obtain a correctly-formatted output for C; (iii) extending the verifier, to account for the additional

round in the transcript; and (iv) extending the simulator, to account for the different ideal functionality (i.e., $f^{\mathsf{S}}_{n,C,\mathcal{G}}$ instead of $f^{\mathsf{E}}_{\tilde{C},\mathcal{G}}$), whose output excludes the aforementioned auxiliary values (which, hence, must be simulated).

Technically, most of the effort goes into constructing $\tilde{C}$ and the simulator of Π^{S}. We thus briefly discuss these two.

The circuit $\tilde{C}$. The circuit $\tilde{C}$ must compute C's output from n multiplicative shares of C's input (chosen at random). If this were the only requirement, then we could simply set $\tilde{C}$ equal to the circuit that, given as input n shares $\vec{\alpha}^{(1)}, \dots, \vec{\alpha}^{(n)} \in \mathbb{F}^m$, first combines the shares into $\vec{\alpha} := (\prod_{j=1}^{n} \alpha_1^{(j)}, \dots, \prod_{j=1}^{n} \alpha_m^{(j)}) \in \mathbb{F}^m$ and then computes $C(\vec{\alpha})$. Unfortunately, such a circuit is not in the class $\mathbf{C}^{\mathsf{E}}$, and thus we cannot invoke Lemma 3.5.3 to securely evaluate $\tilde{C}$ (nor do we know how to obtain an efficient protocol that does). The difficulty thus lies in constructing a circuit $\tilde{C}$ that computes the same function (perhaps with some additional, though simulatable, outputs) and that, moreover, is in $\mathbf{C}^{\mathsf{E}}$.

We thus take an alternative approach, which leverages the fact that C lies in the class $\mathbf{C}^{\mathsf{S}}$. Intuitively, instead of combining shares at the beginning, $\tilde{C}$ combines shares "on the fly", as the circuit is computed, as we now describe. We first focus on the simple case where C has no addition gates, i.e., all gates are either multiplication gates or constant gates. The circuit produced by our approach might seem more complex than necessary; to motivate our approach we thus consider a straight-forward "failed attempt".

Attempt: duplicate and multiply. Consider the following approach for constructing $\tilde{C}$: have $\tilde{C}$ contain n copies of C as a subcircuit (one for each party) such that the $\#\mathsf{inputs}(C)$ inputs of each copy are assigned to a separate input slot of $\tilde{C}$; corresponding outputs of each copy are then multiplied together, thereby combining the shares, via $(n-1) \cdot \#\mathsf{outputs}(C)$ auxiliary multiplication gates. See Figure 3.2 for an example of this approach. Alas, natural extensions of this approach for handling both addition and multiplication gates do not preserve zero-knowledge. See Section 3.10 for detailed discussion.

Our approach. The "failed approach" above can be salvaged by using a slightly different circuit for combining n shares from the individual copies of C. That is, to circumvent issues in simulation we require that outputs of $n-1$ copies of C are not outputs of $\tilde{C}$; such "hiding" modification is zero-knowledge, but not sound. To this end, we use $2(n-1)$-gate subcircuit to multiply the n shares together, which, as compared to $n-1$-gate subcircuit used above, produces $n-1$ additional auxiliary outputs. By carefully choosing the topology of this combining subcircuit we ensure that all of its output wires can be simulated, and that auxiliary outputs suffice to check multiplications by the "hidden" wires. See Figure 3.3a for an example of this approach.

More generally, of course, C may include addition gates and, in such a case, the reduction is more complex, because merely individually evaluating n copies of C and then combining corresponding outputs does not compute the correct function. The reason is not surprising: while multiplicative sharing of inputs commutes with multiplication, it does not commute with addition, and thus it is hard to obtain multiplicative shares of the result of an addition. To circumvent this problem, we break the circuit down into components "separated" by additions, and apply the above idea separately to each. In-between components, before each addition, we combine shares.

In more detail, our construction works as follows. For each maximal subcircuit C_{muls} of C consisting solely of multiplication gates and constant gates, we add n copies of C_{muls} to $\tilde{C}$ and assign each copy to a party, who is responsible to evaluate the copy on his shares of the input. We require that gate output wires in copy assigned to party 1 also be be outputs of $\tilde{C}$ and that output wires from $n-1$ remaining copies of C be internal wires of $\tilde{C}$.

For each multiplication gate g of C we identify the n corresponding copies $g_1, \ldots, g_n$ in $\tilde{C}$ and add a subcircuit C_g that computes the product $\prod_{i=1}^{n} \mathsf{output}(g_i)$, via $2(n-1)+1$ multiplication gates; since inputs are multiplicatively shared, the (main) output of C_g equals the output of g. Finally, for each addition gate g of C we add one addition gate $\tilde{g}$ to $\tilde{C}$ with the following topological modification: whenever g references an output of a multiplication gate g' in C, we set $\tilde{g}$ to reference the output of the corresponding subcircuit $C_{g'}$. The construction ensures that the outputs of multiplication gates in C match the outputs of corresponding subcircuits in $\tilde{C}$; similarly, the outputs of addition gates in C match the outputs of corresponding addition gates in $\tilde{C}$. See Figure 3.3b for an example.

A notable efficiency feature of our reduction is that it ensures that the E-depth of $\tilde{C}$, which determines the number of rounds required to securely evaluate $\tilde{C}$, is "small": it is bounded above by n times the S-depth of C. Indeed, there are multiple ways to combine the aforementioned subcircuits, but many such ways yield much worse efficiency, e.g., E-depth that is as worse as n times the (standard) depth of C. Since the circuits C that we encounter in this paper's application have a small S-depth, this feature is critical.

The simulator in Π^{S}. The construction of $\tilde{C}$ must not only respect syntactic and efficiency requirements (e.g., lie in $\mathbf{C}^{\mathsf{E}}$, not have more than $n \cdot \mathsf{size}(C)$ gates, and so on), but must also be secure, in the sense that the ideal functionality $f^{\mathsf{E}}_{\tilde{C},\mathcal{G}}$ implemented by the evaluation protocol Π^{E} for $\tilde{C}$ actually gives rise (with some simple changes) to a sampling protocol Π^{S} that implements the ideal functionality $f^{\mathsf{S}}_{n,C,\mathcal{G}}$. Since our construction of $\tilde{C}$ introduces additional, spurious outputs, the simulator in Π^{S} must be able to reproduce the view of the adversary when only having access to C's output (rather than $\tilde{C}$'s output). Intuitively, this requires showing that partial computations that carry information about a subset of the parties' shares do not leak additional information beyond the outputs that incorporate every party's share.

For an arbitrary circuit in $\mathbf{C}^{\mathsf{E}}$ such an argument cannot be carried out. However, for the particular circuit $\tilde{C}$ that is constructed from C we show that it is possible to "back compute" the circuit: given the output of C, the simulator can complete it into an output of $\tilde{C}$ by sampling an assignment to the remaining (spurious) output wires of $\tilde{C}$, such that the simulated output is indistinguishable from an evaluation of $\tilde{C}$. This is done by taking each subcircuit in $\tilde{C}$ and computing backwards from its output.

3.5.2 Sketch of the evaluation protocol

We sketch the proof of Lemma 3.5.3. At a high level, the evaluation protocol $\Pi^{\mathsf{E}} = (\Pi, V, S)$ for the circuit class $\mathbf{C}^{\mathsf{E}}$ over a group $\mathbb{G}$ of order r works as follows. Fix a number of parties n and a circuit C in $\mathbf{C}^{\mathsf{E}}$.

- In the first round (i.e., $t = 1$):

  Each party i individually commits to each one of his own private inputs, i.e., each party i commits to the values assigned to wires in $\mathsf{inputs}(C, i)$, and proves, in zero knowledge, knowledge of the committed values; the NP relation used in this case is $\mathcal{R}_{\mathsf{A}}$ in Figure 3.4. Moreover, each party broadcasts the commitments and the NIZK proofs.

- In each one of the subsequent $\mathsf{depth}_{\mathsf{E}}(C)$ rounds (i.e., $t = 2, \ldots, \mathsf{depth}_{\mathsf{E}}(C) + 1$):

  Each party i determines if there are any gates g in $\mathsf{gates}(C)$ such that

  (i) the E-depth of $\mathsf{output}(g)$ equals the round number minus 1 (i.e., $t - 1$), and
  (ii) i is in the dependency set for $\mathsf{output}(g)$ (i.e., $i \in \mathsf{ds}(\mathsf{output}(g))$).

  If so, then party i computes an encoding of the output of each such gate, taken in topological order, and broadcasts this encoding, along with a zero-knowledge proof of its correctness, if needed. The technical conditions above ensure that party i has all the necessary information to compute $\mathsf{output}(g)$ at round t. There are three cases to consider:

  - Case 1: $\mathsf{type}(g) = \mathsf{const}$.

Say that the gate g computes $\mathsf{w} \leftarrow \alpha$. Party i computes and broadcasts the $\mathbb{G}$-element $\mathcal{R} := \alpha \cdot \mathcal{G}$, which encodes the value of the output wire w. Anyone can verify the correctness of the constant encoding, since the coefficient is public, so no zero-knowledge proof is required.

- Case 2: $\mathsf{type}(g) = \mathsf{add}$.

  Say that the gate g computes $\mathsf{w} \leftarrow \sum_{j=1}^{d} \alpha_j \mathsf{w}_j$. Party i consults broadcast messages (including those the party i computed in this round) to obtain $\mathcal{P}_1, \ldots, \mathcal{P}_d \in \mathbb{G}$ that respectively encode the values of the input wires $\mathsf{w}_1, \ldots, \mathsf{w}_d$; he then computes and broadcasts the $\mathbb{G}$-element $\mathcal{R} := \sum_{j=1}^{d} \alpha_j \cdot \mathcal{P}_j$, which encodes the value of the output wire w. Anyone can verify the correctness of the addition, since the coefficients are public and the addends are part of the transcript, so no zero-knowledge proof is required.
- Case 3: $\mathsf{type}(g) = \mathsf{mul}$.

  Say that the gate g computes $\mathsf{w} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$. The value $\sigma \in \mathbb{F}_r$ of the wire w^{R} depends only on values of wires in $\mathsf{inputs}(C, i)$, because C belongs to $\mathbf{C}^{\mathsf{E}}$. Thus, given an encoding $\mathcal{P} \in \mathbb{G}$ of the value of w^{L}, party i can multiply it by α and σ, to obtain an encoding $\mathcal{R} \in \mathbb{G}$ of the value of w. To prove that the multiplication was carried out consistently, party i produces a zero-knowledge proof for the relation $\mathcal{R}_{\mathsf{B,inp}}$ or $\mathcal{R}_{\mathsf{B,priv}}$ in Figure 3.4, which captures the semantics of correct "right multiplication", depending on whether w^{R} is an input to the circuit (captured by $\mathcal{R}_{\mathsf{B,inp}}$) or not (captured by $\mathcal{R}_{\mathsf{B,priv}}$), as follows:

  * If $\mathsf{w}^{\mathsf{R}} \in \mathsf{inputs}(C, i)$, then instance $\mathbb{x}$ for $\mathcal{R}_{\mathsf{B,inp}}$ is a tuple of the form $(\mathsf{crs}, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$ and a witness $\mathbb{w}$ is a tuple of the form (σ, cr). Membership in $\mathcal{R}_{\mathsf{B,inp}}$ requires that $\mathcal{R} = \alpha\sigma \cdot \mathcal{P}$ and cm is a commitment to σ with randomness cr (relative to the common random string crs for the commitment scheme COMM);
  * If $\mathsf{w}^{\mathsf{R}} \notin \mathsf{inputs}(C, i)$, then instance $\mathbb{x}$ for $\mathcal{R}_{\mathsf{B,priv}}$ is a tuple of the form $(\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2, \alpha)$ and a witness $\mathbb{w}$ is a tuple of the form (σ_1, σ_2). Membership in $\mathcal{R}_{\mathsf{B,priv}}$ requires that $\sigma_2 \mathcal{R} = \alpha\sigma_1 \cdot \mathcal{P}$ and $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are encoding of σ_1 and σ_2, respectively.

In sum, the parties prove correct evaluation of all gates of C, first processing all gates whose outputs have E-depth 1, then all those whose outputs have E-depth 2, and so on.

- After $\mathsf{depth}_{\mathsf{E}}(C)$ such rounds, in the last round (i.e., $t = \mathsf{depth}_{\mathsf{E}}(C) + 2$):

  Party 1 consults the broadcast messages so to gather, and broadcast in a single message, the encoding of the value of every output of C. The purpose of this last round is to construct a syntactically well-formed output of the protocol; tasking party 1 to do so is an arbitrary choice.

Since the circuit C belongs to the circuit class $\mathbf{C}^{\mathsf{E}}$, by definition of $\mathbf{C}^{\mathsf{E}}$, whenever a party i is supposed to prove correct evaluation of a gate g, it can do so: (i) if g is an addition gate, then encodings for g's inputs have been broadcast in previous rounds (or computed by i in this round); and (ii) if g is a multiplication gate, an encoding for g's left input has been broadcast in previous rounds (or computed by i in this round) and i knows the value for its right input. In both cases described above, as well as for computing constant gates, party i can compute an encoding for g's output, and knows a witness to the NP statement that attests to this encoding's correctness. Moreover, note that, again since C belongs to $\mathbf{C}^{\mathsf{E}}$, every gate's output wire is also an output wire of C, so that broadcasting encodings of every gate's output does not leak information beyond what is leaked by the output of the ideal functionality, which is $C(\vec{\sigma}) \cdot \mathcal{G}$.

The transcript of broadcast messages can be checked by a verifier V that ensures that the following three properties hold: (1) input commitments carry valid proofs; (2) for each addition gate, the purported output agrees with the direct evaluation of the linear combination; and (3) for each multiplication gate, the party responsible for that gate has produced a valid proof for its evaluation (based on suitable prior values). These checks ensure that the circuit has been consistently evaluated on the parties' private inputs.

Finally, the transcript can be generated by a simulator S, having access to the encoding of the circuit's output, by simulating each proof of correct evaluation.

A subtle technical note is that since the evaluation protocol is only required for valid inputs of C, i.e., inputs in $\mathsf{valid\text{-}inputs}(C)$, we avoid pathological cases. For example, the choice of valid $\mathsf{valid\text{-}inputs}(C)$ for the circuits C in the class $\mathbf{C}^{\mathsf{E}}$ ensures that σ_2 exists in the second sub-case of the Case 3 above.

3.6 Instantiations and optimizations

We discuss how to instantiate and optimize the construction in the proof of Theorem 3.5.1. The sampling protocol of Theorem 3.5.1 is obtained in two steps: a reduction from sampling to evaluation (Lemma 3.5.2), and an evaluation protocol (Lemma 3.5.3). The reduction is explicit and efficient, so we focus on instantiating and optimizing the evaluation protocol, by suitably instantiating the commitment scheme COMM and NIZKs.

In short, we instantiate COMM via Pedersen commitments [Ped92], and the NIZKs via

The NP **relation** $\mathcal{R}_{\mathsf{A}}$**.** An instance-witness pair $(\mathbb{x}, \mathbb{w})$ is in $\mathcal{R}_{\mathsf{A}}$ if and only if all the following checks pass.

1. Parse $\mathbb{x}$ as tuple $(\mathsf{crs}_\star, \mathsf{cm})$ for which $\mathsf{crs}_\star$ is a common random string for COMM and cm is a commitment.
2. Parse $\mathbb{w}$ as a tuple (σ, cr) for which σ is an element in $\mathbb{F}_r$ and cr is commitment randomness.
3. Check that $\mathsf{COMM.V}(\mathsf{crs}_\star, \sigma, \mathsf{cm}, \mathsf{cr}) = 1$.

The NP **relation** $\mathcal{R}_{\mathsf{B,inp}}$**.** An instance-witness pair $(\mathbb{x}, \mathbb{w})$ is in $\mathcal{R}_{\mathsf{B,inp}}$ if and only if all the following checks pass.

1. Parse $\mathbb{x}$ as tuple $(\mathsf{crs}_\star, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$ for which $\mathsf{crs}_\star$ is a common random string for COMM, cm is a commitment, $\mathcal{R}$ and $\mathcal{P}$ are elements in $\mathbb{G}$, and α is an element in $\mathbb{F}_r$.
2. Parse $\mathbb{w}$ as a tuple (σ, cr) for which σ is an element in $\mathbb{F}_r$ and cr is commitment randomness.
3. Check that $\mathcal{R} = \alpha\sigma \cdot \mathcal{P}$.
4. Check that $\mathsf{COMM.V}(\mathsf{crs}_\star, \sigma, \mathsf{cm}, \mathsf{cr}) = 1$.

The NP **relation** $\mathcal{R}_{\mathsf{B,priv}}$**.** An instance-witness pair $(\mathbb{x}, \mathbb{w})$ is in $\mathcal{R}_{\mathsf{B,priv}}$ if and only if all the following checks pass.

1. Parse $\mathbb{x}$ as tuple $(\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2, \alpha)$ for which $\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2$ are elements in $\mathbb{G}$ and α is an element in $\mathbb{F}_r$.
2. Parse $\mathbb{w}$ as a tuple (σ_1, σ_2) for which σ_1 and σ_2 are elements in $\mathbb{F}_r$.
3. Check that $\mathcal{Q}_1 = \sigma_1 \cdot \mathcal{G}$ and $\mathcal{Q}_2 = \sigma_2 \cdot \mathcal{G}$.
4. Check that $\sigma_2\mathcal{R} = \alpha\sigma_1 \cdot \mathcal{P}$ (in particular, that $\mathcal{R} = \alpha\frac{\sigma_1}{\sigma_2} \cdot \mathcal{P}$ if $\sigma_2 \neq 0$).

Figure 3.4: Description of the three NP relations $\mathcal{R}_{\mathsf{A}}, \mathcal{R}_{\mathsf{B,inp}}, \mathcal{R}_{\mathsf{B,priv}}$.

Σ-protocols to which we apply the Fiat–Shamir heuristic [FS87]. The use of NIZKs in the evaluation protocol is "light": the evaluation protocol uses NIZKs for three NP relations that involve only arithmetic in $\mathbb{G}$ and invocations of the commitment verifier COMM.V; we denote these three relations by $\mathcal{R}_{\mathsf{A}}, \mathcal{R}_{\mathsf{B,inp}}, \mathcal{R}_{\mathsf{B,priv}}$ and define them in Figure 3.4. Moreover, the theorem relies on the NIZK proof of knowledge only when proving statements relative to $\mathcal{R}_{\mathsf{A}}$, but not when proving statements relative to $\mathcal{R}_{\mathsf{B,inp}}$ or $\mathcal{R}_{\mathsf{B,priv}}$. We instantiate the NIZKs by relying on a random oracle $\mathcal{H}$ that maps $\{0,1\}^*$ to $\mathbb{F}_r$ (which then replaces the NIZK common random strings as a setup assumption) because we apply the Fiat–Shamir heuristic [FS87] to certain Σ-protocols, discussed further below. In our code implementation we heuristically instantiate $\mathcal{H}$ via the `SHA256` hash function. We now describe the instantiation of COMM and those of the NIZKs for the three relations.

Choice of the commitment scheme. We use Pedersen commitments [Ped92] to instantiate COMM. The common random string $\mathsf{crs}_\star$ consists of two generators of $\mathbb{G}$ for which there is no known linear relation: $\mathsf{crs}_\star := (\mathcal{G}, \mathcal{G}')$. If $\mathbb{G}$ is a prime-order elliptic curve group, $\mathcal{G}$ and $\mathcal{G}'$ can be found by applying point decompression to two random strings or, heuristically, to `SHA256`(0) and `SHA256`(1). The generator COMM.G and verifier COMM.V work as follows.

- $\mathsf{COMM.G}(\mathsf{crs}_\star, \sigma) \to (\mathsf{cm}, \mathsf{cr})$:
  1. Parse the common random string $\mathsf{crs}_\star$ as a pair $(\mathcal{G}, \mathcal{G}')$.
  2. Sample the commitment randomness cr to be a random element of $\mathbb{F}_r$.
  3. Compute the commitment cm as $\mathsf{cm} := \sigma \cdot \mathcal{G} + \mathsf{cr} \cdot \mathcal{G}' \in \mathbb{G}$.
  4. Return $(\mathsf{cm}, \mathsf{cr})$.
- $\mathsf{COMM.V}(\mathsf{crs}_\star, \sigma, \mathsf{cm}, \mathsf{cr}) \to b$:
  1. Parse the common random string $\mathsf{crs}_\star$ as a pair $(\mathcal{G}, \mathcal{G}')$.
  2. Compute $\widehat{\mathsf{cm}} := \sigma \cdot \mathcal{G} + \mathsf{cr} \cdot \mathcal{G}' \in \mathbb{G}$.
  3. If $\mathsf{cm} = \widehat{\mathsf{cm}}$, return $b := 1$; otherwise, return $b := 0$.

The value of $\mathsf{cr} \cdot \mathcal{G}'$ is uniformly random in $\mathbb{G}$, so cm is statistically hiding. Given two decommitments and corresponding commitment randomnesses $(\sigma_1, \mathsf{cr}_1)$ and $(\sigma_2, \mathsf{cr}_1)$ with $\sigma_1 \neq \sigma_2$ one can compute $\log_{\mathcal{G}} \mathcal{G}' := (\mathsf{cr}_2 - \mathsf{cr}_1)/(\sigma_2 - \sigma_1)$. Therefore cm is also computationally binding given the hardness of finding discrete logarithms in $\mathbb{G}$.

Choice of NIZK for the relation $\mathcal{R}_{\mathrm{A}}$. The relation $\mathcal{R}_{\mathrm{A}}$ captures the semantics of knowing the value σ hidden in a commitment cm for the scheme COMM. An instance $\mathbb{x}$ has the form $(\mathsf{crs}_\star, \mathsf{cm})$, while a witness $\mathbb{w}$ has the form (σ, cr); a pair $(\mathbb{x}, \mathbb{w})$ is in $\mathcal{R}_{\mathrm{A}}$ if and only if $\mathsf{COMM.V}(\mathsf{crs}_\star, \sigma, \mathsf{cm}, \mathsf{cr}) = 1$. Given that COMM is instantiated via Pedersen commitments (see above), we use an adapted version of Schnorr's protocol for proving knowledge of discrete logarithms [Sch91] to obtain a NIZK (of knowledge) for $\mathcal{R}_{\mathrm{A}}$. Below, we directly describe the non-interactive protocol obtained after applying the Fiat–Shamir heuristic based on a random oracle $\mathcal{H}$. The prover $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}.\mathsf{P}$ and verifier $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}.\mathsf{V}$ work as follows.

- $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}.\mathsf{P}(\mathcal{H}, \mathbb{x}, \mathbb{w}) \to \pi$:
  1. Parse the instance $\mathbb{x}$ as $(\mathsf{crs}_\star, \mathsf{cm})$ and $\mathsf{crs}_\star$ as $(\mathcal{G}, \mathcal{G}')$, and parse the witness $\mathbb{w}$ as (σ, cr).
  2. Sample the announcement nonces $\gamma, \delta \in \mathbb{F}_r$ at random.
  3. Compute the *announcement* $\mathcal{R}$ as $\mathcal{R} := \gamma \cdot \mathcal{G} + \delta \cdot \mathcal{G}' \in \mathbb{G}$.
  4. Compute the *challenge* c as $c := \mathcal{H}(\mathsf{crs}_\star \| \mathsf{cm} \| \mathcal{R}) \in \mathbb{F}_r$.
  5. Compute the *response* (μ, ν) as $\mu := \gamma + c \cdot \sigma \in \mathbb{F}_r$ and $\nu := \delta + c \cdot \mathsf{cr} \in \mathbb{F}_r$.
  6. Return $\pi := (\mathcal{R}, \mu, \nu)$.
- $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}.\mathsf{V}(\mathcal{H}, \mathbb{x}, \pi) \to b$:
  1. Parse the instance $\mathbb{x}$ as $(\mathsf{crs}_\star, \mathsf{cm})$ and $\mathsf{crs}_\star$ as $(\mathcal{G}, \mathcal{G}')$, and parse the proof π as $(\mathcal{R}, \mu, \nu)$.
  2. Compute the challenge c as $c := \mathcal{H}(\mathsf{crs}_\star \| \mathsf{cm} \| \mathcal{R}) \in \mathbb{F}_r$.
  3. Check if $\mu \cdot \mathcal{G} + \nu \cdot \mathcal{G}' = \mathcal{R} + c \cdot \mathsf{cm}$. If so, return $b := 1$; otherwise, return $b := 0$.

One can check that the above construction is a zero knowledge proof of knowledge in the random oracle model.

Choice of NIZK for the relation $\mathcal{R}_{\mathsf{B,inp}}$. The relation $\mathcal{R}_{\mathsf{B,inp}}$ captures the semantics of correct evaluation of a "multiplication-by-input" gate, i.e., of correctly multiplying an encoding $\mathcal{P}$ of a wire value by a (plain) wire value σ to obtain the encoding $\mathcal{R}$ of the output wire. An instance $\mathbb{x}$ has the form $(\mathsf{crs}_\star, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$, while a witness $\mathbb{w}$ has the form (σ, cr). We have that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\mathsf{B,inp}}$ if and only if $\mathcal{R} = \alpha\sigma \cdot \mathcal{P}$ and $\mathsf{COMM.V}(\mathsf{crs}_\star, \sigma, \mathsf{cm}, \mathsf{cr}) = 1$.

To implement this relation we use equality composition [CP92] of the following two Σ-protocols: (i) the protocol for knowledge of a Pedersen commitment outlined above; and (ii) Schnorr's protocol for knowledge of an "implicit" discrete logarithm. As before, we directly describe the non-interactive protocol obtained after applying the Fiat–Shamir heuristic based on a random oracle $\mathcal{H}$. The prover $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{P}$ and verifier $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{V}$ work as follows.

- $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{P}(\mathcal{H}, \mathbb{x}, \mathbb{w}) \to \pi$:
  1. Parse the instance $\mathbb{x}$ as $(\mathsf{crs}_\star, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$ and $\mathsf{crs}_\star$ as $(\mathcal{G}, \mathcal{G}')$, and parse the witness $\mathbb{w}$ as (σ, cr).
  2. Sample the announcement nonces $\gamma, \delta \in \mathbb{F}_r$ at random.
  3. Compute the *announcement* $(\mathcal{X}, \mathcal{Y})$ as $\mathcal{X} := \gamma \cdot \mathcal{G} + \delta \cdot \mathcal{G}' \in \mathbb{G}$ and $\mathcal{Y} := \alpha\gamma \cdot \mathcal{P} \in \mathbb{G}$.
  4. Compute the *challenge* c as $c := \mathcal{H}(\mathsf{crs}_\star \| \mathsf{cm} \| \mathcal{X} \| \mathcal{Y}) \in \mathbb{F}_r$.
  5. Compute the *response* (u, v) as $u := \gamma \cdot c + \sigma \in \mathbb{F}_r$ and $v := \delta \cdot c + \mathsf{cr} \in \mathbb{F}_r$.
  6. Return $\pi := (\mathcal{X}, \mathcal{Y}, u, v)$.
- $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{V}(\mathcal{H}, \mathbb{x}, \pi) \to b$:
  1. Parse the instance $\mathbb{x}$ as $(\mathsf{crs}_\star, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$ and $\mathsf{crs}_\star$ as $(\mathcal{G}, \mathcal{G}')$.
  2. Parse the proof π as $(\mathcal{X}, \mathcal{Y}, u, v)$.
  3. Compute the challenge c as $c := \mathcal{H}(\mathsf{crs}_\star \| \mathsf{cm} \| \mathcal{X} \| \mathcal{Y}) \in \mathbb{F}_r$.
  4. Check if $u \cdot \mathcal{G} + v \cdot \mathcal{G}' = c \cdot \mathcal{X} + \mathsf{cm}$ and $\alpha u \cdot \mathcal{P} = c \cdot \mathcal{Y} + \mathcal{R}$. If so, return $b := 1$; otherwise, return $b := 0$.

One can check that the above construction is a zero knowledge proof in the random oracle model; also, we do not need the construction to also be a proof of knowledge.

Choice of NIZK for the relation $\mathcal{R}_{\mathsf{B,priv}}$. The relation $\mathcal{R}_{\mathsf{B,priv}}$ captures the semantics of correct evaluation of a "multiplication-by-non-input" gate, i.e., of correctly multiplying an encoding $\mathcal{P}$ of a wire value by a (plain) wire value σ to obtain the encoding $\mathcal{R}$ of the output wire. The circuit membership in $\mathbf{C}^{\mathsf{E}}$ means that even if σ is not carried by a circuit output wire, there do exist two output wires whose values σ_1 and σ_2 have ratio of $\sigma = \frac{\sigma_1}{\sigma_2}$. An instance $\mathbb{x}$ has the form $(\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2, \alpha)$, while a witness has the form (σ_1, σ_2) for which $\sigma_1, \sigma_2 \in \mathbb{F}_r$. We have that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\mathsf{B,priv}}$ if and only if $\sigma_2\mathcal{R} = \alpha\sigma_1 \cdot \mathcal{P}$ and $\mathcal{Q}_1 = \sigma_1 \cdot \mathcal{G}$,

$\mathcal{Q}_2 = \sigma_2 \cdot \mathcal{G}$.

In a duplex pairing group (see Section 3.3.5), the proof for this NIZK is just $\perp$, because the relationship captured by $\mathcal{R}_{\mathsf{B,inp}}$ is publicly verifiable: $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\mathsf{B,priv}}$ if and only if $e(\mathcal{Q}_2, \mathcal{R}) = e(\mathcal{Q}_1, \alpha \cdot \mathcal{P})$. This construction is trivially a zero knowledge proof because the proof is empty; also, we do not need the construction to also be a proof of knowledge. (Our implementation and evaluation target the special case of duplex pairing groups, which occurs, e.g., in the setting of public-parameter generation for zk-SNARKs.)

An optimization for repeated multiplications. The three NIZKs described above are relatively lightweight, but our implementation leverages an additional optimization, for duplex pairing groups, that we now describe. If an input wire w participates in more than one multiplication gate, then only one multiplication by the value of w needs to carry a NIZK proof for $\mathcal{R}_{\mathsf{B,inp}}$ and all other such multiplications do not require a proof in order to check their correct evaluation; overall, we only need one NIZK proof for $\mathcal{R}_{\mathsf{B,inp}}$ per input wire (and note that this optimization preserves zero knowledge). Details follow.

Let the input wire w be used in a multiplication gate g computing $g\colon \alpha \cdot \mathsf{w}^{\mathsf{L}} \cdot \mathsf{w} \to \mathsf{w}^{\mathsf{out}}$; let σ be the value of w, cm the commitment to σ, and $\mathcal{P}, \mathcal{R}$ the encodings of the values of $\mathsf{w}^{\mathsf{L}}, \mathsf{w}^{\mathsf{out}}$. To ensure that $\frac{1}{\alpha} \log_{\mathcal{P}} \mathcal{R}$ equals σ hidden in cm, we produce a NIZK proof π for a suitable instance $\mathbb{x}$ of $\mathcal{R}_{\mathsf{B,inp}}$. But now suppose that w is also used in another multiplication gate $\tilde{g}$ computing $\tilde{g}\colon \tilde{\alpha} \cdot \tilde{\mathsf{w}}^{\mathsf{L}} \cdot \mathsf{w} \to \tilde{\mathsf{w}}^{\mathsf{out}}$, and let $\tilde{\mathcal{P}}, \tilde{\mathcal{R}}$ be the encodings of the values of the wires $\tilde{\mathsf{w}}^{\mathsf{L}}, \tilde{\mathsf{w}}^{\mathsf{out}}$. Then, given π, α, $\mathcal{P}$, and $\mathcal{R}$ (all in $\mathbb{x}$), anyone can verify that $\frac{1}{\tilde{\alpha}} \log_{\tilde{\mathcal{P}}} \tilde{\mathcal{R}}$ equals σ hidden in cm by checking that $e(\mathcal{P}, \tilde{\mathcal{R}})^{\alpha} = e(\tilde{\mathcal{P}}, \mathcal{R})^{\tilde{\alpha}}$ (because this implies that $\frac{1}{\tilde{\alpha}} \log_{\tilde{\mathcal{P}}} \tilde{\mathcal{R}}$ equals $\frac{1}{\alpha} \log_{\mathcal{P}} \mathcal{R}$, which equals σ as attested to by π), so that no new proof $\tilde{\pi}$ is required for this second statement.

3.7 Implementation

Our system. We built a system that implements our constructions for duplex pairing groups (see Section 3.3.5). Given a prime r, an order-r duplex-pairing group $\mathbb{G} = \langle \mathcal{G} \rangle$, and an $\mathbb{F}_r$-arithmetic circuit $C\colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ in the class $\mathbf{C}^{\mathsf{S}}$, our system provides a multi-party protocol for securely sampling $C(\vec{\alpha}) \cdot \mathcal{G}$ for random $\vec{\alpha}$ in $\mathbb{F}_r^m$. Specifically, the system instantiates and implements the constructions underlying Section 3.5's theorems, in the case when $\mathbb{G}$ is a duplex-pairing group. Our system comprises 1141 lines of C++.

Application to zk-SNARKs via integration with libsnark. The parameter generator of many zk-SNARK constructions works as follows: evaluate a certain circuit C at a random input $\vec{\alpha}$, and then output $\mathsf{pp} := C(\vec{\alpha}) \cdot \mathcal{G}$ as the proof system's public parameters. (See discussion in Section 3.1.2.) Thus, our system can be used to securely sample public parameters of a zk-SNARK, provided that the circuit used in its generator belongs to the circuit class $\mathbf{C}^{\mathsf{S}}$. To facilitate this application, we have integrated our code with **libsnark** [SCI], a C++ library for zk-SNARKs. (In particular, the sampled pp can be used directly by **libsnark**.)

Two zk-SNARK constructions. We worked out circuits for parameter generation for two (preprocessing) zk-SNARK constructions: the one of [PGHR13, BCTV14c] and the one of [DFGK14]. The first zk-SNARK "natively" supports proving satisfiability of *arithmetic* circuits, while the second zk-SNARK that of *boolean* circuits.[5]

Specifically, we wrote code that produces a circuit $C_{\mathsf{PGHR}} \in \mathbf{C}^{\mathsf{S}}$ that can be used to generate public parameters for [PGHR13, BCTV14c]'s zk-SNARK; likewise for producing a circuit $C_{\mathsf{DFGK}} \in \mathbf{C}^{\mathsf{S}}$ for [DFGK14]'s zk-SNARK. We have invoked our system on both circuit types, and demonstrated the secure sampling of respective public parameters.

See Appendix 3.12 for more information about these examples. A critical issue discussed there is ensuring that C_{PGHR} and C_{DFGK} have size quasilinear in the size of the circuit whose satisfiability is being proved. A naive implementation of the computation pattern of the zk-SNARK's generator results in circuits that are not in $\mathbf{C}^{\mathsf{S}}$; conversely, a naive implementation in $\mathbf{C}^{\mathsf{S}}$ results in circuits of quadratic size. Via careful design, quasilinear-size circuits in $\mathbf{C}^{\mathsf{S}}$ can be obtained.

3.8 Evaluation

We report the evaluation of our system (described in Section 3.7).

Setup. We evaluated our system on a desktop PC with a 3.40 GHz Intel Core i7-4770 CPU and 16 GB of RAM available. All experiments are in single-thread mode (though our code also supports multiple-thread mode).

When invoking functionality from **libsnark** (with which our code is integrated), we need to specify, via a build option, a pairing-friendly elliptic curve, which determines how the duplex-pairing group $\mathbb{G}$ is instantiated. The **libsnark** library offers (among others) the following pairing-friendly elliptic curves:

[5] More precisely, both [PGHR13, BCTV14c] and [DFGK14] actually support more general NP relations (phrased in terms of systems of equations) but, for simplicity, we ignore this technical detail in this and later discussions.

- `BN-128` (a Barreto–Naehrig curve [BN06] at 128 bits of security);
- `MNT4-80` (a Miyaji—Nakabayashi—Takano curve with embedding degree 4 [MNT01] at 80 bits of security); and
- `MNT6-80` (a Miyaji—Nakabayashi—Takano curve with embedding degree 6 [MNT01] at 80 bits of security).

The first option implies that $\mathbb{G}$'s order r is a 256-bit prime, and is our default choice for experiments. The second and third options each imply that r is a 298-bit prime, and are used in one of the zk-SNARK applications (see below).

Costs for the general case. Our system's costs depend on the number n of participating parties, and on the size and S-depth of the circuit C in $\mathbf{C}^{\mathsf{S}}$. In Figure 3.6 we report cost models for several complexity measures: the number of broadcast rounds, each party's time complexity, the number of broadcast messages, the transcript size, and the transcript verification time. (Figure 3.6 also reports costs for two concrete examples, discussed further down below.)

Costs for two zk-SNARK constructions. When applying our system to generate public parameters for a zk-SNARK, we construct a circuit C in $\mathbf{C}^{\mathsf{S}}$ so that $C(\vec{\alpha}) \cdot \mathcal{G}$ (for random $\vec{\alpha}$) equals the zk-SNARK's generator output distribution. This distribution depends on the particular NP relation given as input to the generator; thus, the circuit C also depends on this NP relation. Moreover, different zk-SNARK constructions "natively" support different classes of NP relations.

In order to know our system's efficiency when applied to generate zk-SNARK public parameters, we report the size and S-depth of the circuit C as a function of the input NP relation, relative to two zk-SNARK constructions.

- *The zk-SNARK [DFGK14].* This zk-SNARK supports boolean circuit satisfiability: the generator receives as input a boolean circuit D, and outputs public parameters for proving D's satisfiability. If D has N_{w} wires and N_{g} gates, our code outputs a corresponding circuit $C := C_{\mathsf{DFGK}}$ with size $2N_{\mathrm{w}} + 2^{\lceil \log_2 N_{\mathrm{g}} \rceil}(\lceil \log_2 N_{\mathrm{g}} \rceil + 1) + 10$ and S-depth 2.

- *The zk-SNARK of [PGHR13, BCTV14c].* This zk-SNARK supports arithmetic circuit satisfiability: the generator receives as input an arithmetic circuit D, and outputs public parameters for proving D's satisfiability. If D has N_{w} wires and N_{g} gates, our code outputs a corresponding circuit $C := C_{\mathsf{PGHR}}$ with size $11N_{\mathrm{w}} + 2^{\lceil \log_2 N_{\mathrm{g}} \rceil}(\lceil \log_2 N_{\mathrm{g}} \rceil + 1) + 38$ and S-depth 3.

zk-SNARK	**Circuit satisfiability of D when D is**	**Corresponding circuit C in $\mathbf{C}^{\mathsf{S}}$**		
		$\mathsf{size}(C)$	$\mathsf{depth}(C)$	$\mathsf{depth}_{\mathsf{S}}(C)$
[DFGK14]	a N_{w}-wire N_{g}-gate boolean circuit	$2N_{\mathsf{w}} + 2^{\lceil \log_2 N_{\mathsf{g}} \rceil}(\lceil \log_2 N_{\mathsf{g}} \rceil + 1) + 10$	$XXX : datapoint$	2
[PGHR13, BCTV14c]	a N_{w}-wire N_{g}-gate arithmetic circuit	$11N_{\mathsf{w}} + 2^{\lceil \log_2 N_{\mathsf{g}} \rceil}(\lceil \log_2 N_{\mathsf{g}} \rceil + 1) + 38$	$XXX : datapoint$	3
[BCG$^+$14]	Example #1's arithmetic circuit	138 467 206	$XXX : datapoint$	3
[BCTV14b]	Example #2's arithmetic circuit	8 027 609	$XXX : datapoint$	6

Figure 3.5: Size, depth, and S-depth of the circuit C in $\mathbf{C}^{\mathsf{S}}$ obtained from D, for various choices of D.

Complexity measure	**Cost for**		
	general case (with BN-128)	**Example #1** (with BN-128)	**Example #2** (with MNT4-80 and MNT6-80)
number of broadcast rounds	$n \cdot \mathsf{depth}_{\mathsf{S}}(C) + 3$	$3n + 3$	$6n + 6$
each party's time complexity	$0.035 \cdot \mathsf{size}(C)$ ms	14 124 s	4 048 s
number of broadcast messages	$n \cdot (\mathsf{depth}_{\mathsf{S}}(C) + 3)$	$6n$	$6n$
transcript size	$0.072 \cdot n \cdot \mathsf{size}(C)$ kB	$12\,877 \cdot n$ MB	$906 \cdot n$ MB
transcript verification time	$1.03 \cdot n \cdot \mathsf{size}(C)$ ms	$196\,208 \cdot n$ s	$50\,945 \cdot n$ s

Figure 3.6: Our system's costs for the general case, Example #1, and Example #2; n is the number of parties.

These costs are summarized in Figure 3.5 (alongside two concrete examples, discussed next).

Costs for two concrete examples. We report costs for the following concrete choices of a circuit $C := C_{\mathsf{PGHR}}$.

- **Example #1:** the circuit C targets Zerocash [BCG$^+$14]. Namely, $C(\vec{\alpha}) \cdot \mathcal{G}$ (for random $\vec{\alpha}$) equals the output distribution of the generator of the preprocessing zk-SNARK on which Zerocash is based. We selected this example because [BCG$^+$14] 's authors had acknowledged the need, in practice, to securely sample Zerocash's parameters.

- **Example #2:** the circuit C targets the scalable zk-SNARK of [BCTV14b]. Namely, $C(\vec{\alpha}) \cdot \mathcal{G}$ (for random $\vec{\alpha}$) equals the output distribution of the generator used to set up the scalable zk-SNARK. We selected this example because, in this case, the generator's output is *universal* (it suffices for proving *any* computation expressed as machine code on a certain RISC machine), so that the zk-SNARK's parameters can be securely sampled once and for all.

Figure 3.5 reports the size, depth, and S-depth of C for these two examples, and Figure 3.6 reports the corresponding costs of our system when run on these choices of C.

3.9 Conclusion

Like time and space, trust is also a costly resource. To facilitate the deployment of NIZKs and, in particular, zk-SNARKs in various applications, it is not only important to minimize the time and space requirements of proving and verification, but also the trust requirements of parameter generation.

The system that we have presented in this paper can be used to reduce the trust requirements of parameter generation for a class of zk-SNARKs: the system provides a multi-party broadcast protocol in which only one honest party, out of n participating ones, is required to securely sample the public parameters. Integration of our system with **libsnark** greatly facilitates this application. As a demonstration, we have used our system for securely sampling public parameters for the zk-SNARKs of [PGHR13, BCTV14c, DFGK14].

Several questions remain open. First, can our multi-party protocol (or a modification of it) be proved secure against adaptive corruptions? Our analysis considered only non-adaptive corruptions.

Next, can one efficiently support secure sampling for circuits C outside the class $\mathbf{C}^{\mathsf{S}}$ (e.g., with division gates)? Supporting a larger circuit class $\mathbf{C}$ may let us (i) express the generators of [PGHR13, BCTV14c, DFGK14] via circuits C in $\mathbf{C}$ with smaller costs, and (ii) express the generators of additional zk-SNARK constructions via circuits C in $\mathbf{C}$.

Finally, in this work we have not attempted to tackle the "human component" of parameter generation. Namely, once we have a system that allows secure sampling via a multi-party protocol, how should we choose the participating parties? What penalties should be put in place for misbehavior, if any? Where and how should the protocol be conducted? These questions, too, need good answers in order to convincingly sample public parameters via the multi-party protocol.

3.10 Proof of Lemma 3.5.2

We prove Lemma 3.5.2. Specifically, first we describe the construction of the circuit transformation T_1, and then the construction of the protocol transformation T_2; afterwards, we explain why these constructions work.

Construction of T_1. On input a positive integer n and a circuit $C \colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ in the class $\mathbf{C}^{\mathsf{S}}$, the transformation T_1 outputs a circuit $\tilde{C}$ in the class $\mathbf{C}^{\mathsf{E}}$.

First we describe high-level properties of the circuit $\tilde{C}$. The number of wires, gates, inputs, and outputs of $\tilde{C}$ are at most a multiplicative factor of $O(n)$ larger than those of C:

- $\#\mathsf{wires}(\tilde{C}) \leq (3n-2) \cdot \#\mathsf{wires}(C) + 1$,
- $\#\mathsf{gates}(\tilde{C}) = \#\mathsf{const\text{-}gates}(C) + \#\mathsf{add\text{-}gates}(C) + (3n-2) \cdot \#\mathsf{mul\text{-}gates}(C) + 1 \leq (3n-2) \cdot \#\mathsf{gates}(C) + 1$,
- $\#\mathsf{inputs}(\tilde{C}) = n \cdot \#\mathsf{inputs}(C)$, and
- $\#\mathsf{outputs}(\tilde{C}) \leq (2n-1) \cdot \#\mathsf{outputs}(C) + 1$.

The inputs of $\tilde{C}$ are partitioned into n slots each of size m and, for each i, $\mathsf{size}(\tilde{C}, i) = O(\mathsf{size}(C))$. In particular, we can write $\tilde{C} \colon \mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n} \to \mathbb{F}^{nh}$ with each m_i equal to m. The E-depth of $\tilde{C}$ is n times the S-depth of C: $\mathsf{depth}_{\mathsf{E}}(\tilde{C}) = n \cdot \mathsf{depth}_{\mathsf{S}}(C)$.

Moreover, there is a *wire embedding* from C to $\tilde{C}$, i.e., a map $\phi\colon \mathsf{outputs}(C) \to \mathsf{outputs}(\tilde{C})$ that works as follows. Consider any $\vec{\alpha}^{(1)}, \dots, \vec{\alpha}^{(n)} \in \mathbb{F}^m$ and let $\vec{\alpha} := (\prod_{j=1}^{n} \alpha_1^{(j)}, \dots, \prod_{j=1}^{n} \alpha_m^{(j)}) \in \mathbb{F}^m$. Then, for every $\mathsf{w} \in \mathsf{outputs}(C)$, the value assigned to w when computing $C(\vec{\alpha})$ equals the value assigned to $\phi(\mathsf{w}) \in \mathsf{outputs}(\tilde{C})$ when computing $\tilde{C}(\vec{\alpha}^{(1)}, \dots, \vec{\alpha}^{(n)})$. In other words, if $\tilde{C}$'s input corresponds to a multiplicative sharing, among n parties, of C's input, then $\tilde{C}$'s output contains C's output (and other values), and ϕ specifies the embedding from the latter into the former.

We now turn to the construction of the circuit $\tilde{C}$ from C. For notational convenience, we assume that C's wires have a topological order: for every wire $\mathsf{w} \in \mathsf{wires}(C)$, $\mathsf{idx}(\mathsf{w})$ denotes the *index* of w according to this order; also, C's input wires have indexes from 1 to m. Initialize $\tilde{C}$ to be an empty circuit with domain $\mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n}$, and denote by $\tilde{\mathsf{w}}_{i,k}$ the i-th input wire of $\tilde{C}$'s k-th input slot (i ranges from 1 to m and k ranges from 1 to n). The procedure described below iteratively adds gates and wires to $\tilde{C}$ by considering in turn each gate of C. It also builds a size $\#\mathsf{outputs}(C)$ vector $\vec{\phi}$ representing the wire embedding ϕ.

Roughly, the circuit $\tilde{C}$ has the following structure:

- n copies of the sampling circuit C; each copy is assigned to a party tasked to evaluate C on its shares of the input. In particular, for each wire w_i of C, the circuit $\tilde{C}$ has n wires $\tilde{\mathsf{w}}_{i,1}, \dots, \tilde{\mathsf{w}}_{i,n}$ whose values correspond to multiplicative shares of w_i's value.
- For each wire w_i of C that is an output of a multiplication gate, the circuit $\tilde{C}$ has $2n-2$ gates $\tilde{g}_{i,2}^{\mathsf{aux}}, \dots, \tilde{g}_{i,n}^{\mathsf{aux}}$ and $\tilde{g}_{i,2}^{\mathsf{out}}, \dots, \tilde{g}_{i,n}^{\mathsf{out}}$ with corresponding output wires $\tilde{\mathsf{w}}_{i,2}^{\mathsf{aux}}, \dots, \tilde{\mathsf{w}}_{i,n}^{\mathsf{aux}}$ and $\tilde{\mathsf{w}}_{i,2}^{\mathsf{out}}, \dots, \tilde{\mathsf{w}}_{i,n}^{\mathsf{out}}$ tasked with multiplying the aforementioned shares together[6]. In this

[6] One can multiply n shares with a size $n-1$ circuit. Unfortunately, this approach yields a construction that cannot be simulated unless the CDH assumption is false. Our approach achieves zero-knowledge and requires just a slight increase in circuit size.

subcircuit of $\tilde{C}$, the value of $\tilde{\mathsf{w}}_{i,n}^{\mathsf{out}}$ equals that of w_i in C.

In more detail, the circuit $\tilde{C}$ and wire embedding ϕ are constructed as follows. Let ϕ be a vector of size $\#\mathsf{inputs}(C)$, initially having all its entries set to $\perp$. Let $\tilde{C}$ be a circuit with a partitioned domain $C\colon \prod_{i=1}^{n} \mathbb{F}^{\#\mathsf{inputs}(C)} \to \mathbb{F}^h$ and denote its inputs as $\tilde{\mathsf{w}}_{i,j}$, where $i = 1, \ldots, \#\mathsf{inputs}(C)$ and $j = 1, \ldots, n$. Extend $\tilde{C}$ with a constant gate outputting 1 and denote the corresponding output wire by $\tilde{\mathsf{w}}_0$. This "dummy" gate is used to provide 1 as a multiplicative share of certain values (and in many instances can be elided in the final circuit).

Finally, for each non-input wire w of C (i.e., in $\mathsf{wires}(C) \setminus \mathsf{inputs}(C)$) taken in topological order, do the following.

1. Set $i := \mathsf{idx}(\mathsf{w})$.
2. Letting g be the gate whose output wire is w, we distinguish between several cases depending on g.
   - g is a constant gate computing $\mathsf{w} \leftarrow \alpha$
     (a) Extend $\tilde{C}$ with a new constant gate $\tilde{g}_i$ and a corresponding output wire $\tilde{\mathsf{w}}_i$.
     (b) Have $\tilde{g}_i$ compute $\tilde{\mathsf{w}}_i \leftarrow \alpha$ and have $\tilde{\mathsf{w}}_i$ be an output of $\tilde{C}$.
     (c) Set $\phi(\mathsf{w}) := \tilde{\mathsf{w}}_i$.
     (d) Henceforth treat all references to $\tilde{\mathsf{w}}_{i,1}$ as references to $\tilde{\mathsf{w}}_i$ and, for $k = 2, \ldots, n$, treat all references to $\tilde{\mathsf{w}}_{i,k}$ as references to $\tilde{\mathsf{w}}_0$. Moreover, treat all references to $\tilde{\mathsf{w}}_{i,k}^{\mathsf{out}}$ $(k = 1, \ldots, n)$ as references to $\tilde{\mathsf{w}}_i$.
   - g is an addition gate computing $\mathsf{w} \leftarrow \sum_{j=1}^{d} \alpha_j \mathsf{w}_j$
     (a) Extend $\tilde{C}$ with a new addition gate $\tilde{g}_i$ and a corresponding output wire $\tilde{\mathsf{w}}_i$.
     (b) Have $\tilde{g}_i$ compute $\tilde{\mathsf{w}}_i \leftarrow \sum_{j=1}^{d} \alpha_j \phi(\mathsf{w}_j)$ and have $\tilde{\mathsf{w}}_i$ be an output of $\tilde{C}$.
     (c) Henceforth treat all references to $\tilde{\mathsf{w}}_{i,1}$ as references to $\tilde{\mathsf{w}}_i$ and, for $k = 2, \ldots, n$, treat all references to $\tilde{\mathsf{w}}_{i,k}$ as references to $\tilde{\mathsf{w}}_0$. Moreover, treat all references to $\tilde{\mathsf{w}}_{i,k}^{\mathsf{out}}$ $(k = 1, \ldots, n)$ as references to $\tilde{\mathsf{w}}_i$.
     (d) Set $\phi(\mathsf{w}) := \tilde{\mathsf{w}}_i$.
   - g is a multiplication gate computing $\mathsf{w} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$
     (a) Set $l := \mathsf{idx}(\mathsf{w}^{\mathsf{L}})$ and $r := \mathsf{idx}(\mathsf{w}^{\mathsf{R}})$.
     (b) Extend $\tilde{C}$ with n new multiplication gates $\tilde{g}_{i,1}, \ldots, \tilde{g}_{i,n}$ and n corresponding output wires $\tilde{\mathsf{w}}_{i,1}, \ldots, \tilde{\mathsf{w}}_{i,n}$.
     (c) Have $\tilde{g}_{i,1}$ compute $\tilde{\mathsf{w}}_{i,1} \leftarrow \alpha \cdot \tilde{\mathsf{w}}_{l,1} \cdot \tilde{\mathsf{w}}_{r,1}$ and, for $k = 2, \ldots, n$, have $\tilde{g}_{i,k}$ compute $\tilde{\mathsf{w}}_{i,k} \leftarrow 1 \cdot \tilde{\mathsf{w}}_{l,k} \cdot \tilde{\mathsf{w}}_{r,k}$. Have $\tilde{\mathsf{w}}_{i,1}$ be an output of $\tilde{C}$ and have $\tilde{\mathsf{w}}_{i,2}, \ldots, \tilde{\mathsf{w}}_{i,n}$ be internal wires of $\tilde{C}$.

(d) Henceforth treat all references to $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,1}$ as references to $\tilde{\mathsf{w}}_{i,1}$.

(e) Extend $\tilde{C}$ with $2n-2$ new multiplication gates $\tilde{g}^{\mathsf{out}}_{i,2}, \ldots, \tilde{g}^{\mathsf{out}}_{i,n}$ and $\tilde{g}^{\mathsf{aux}}_{i,2}, \ldots, \tilde{g}^{\mathsf{aux}}_{i,n}$, as well as $2n-2$ corresponding output wires $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,2}, \ldots, \tilde{\mathsf{w}}^{\mathsf{out}}_{i,n}$ and $\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,2}, \ldots, \tilde{\mathsf{w}}^{\mathsf{aux}}_{i,n}$.

(f) For $k = 2, \ldots, n$ have $\tilde{g}^{\mathsf{aux}}_{i,k}$ compute $\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k} \leftarrow 1 \cdot \tilde{\mathsf{w}}^{\mathsf{out}}_{i,k-1} \cdot \tilde{\mathsf{w}}_{l,k}$ and have $\tilde{g}^{\mathsf{out}}_{i,k}$ compute $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k} \leftarrow 1 \cdot \tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k} \cdot \tilde{\mathsf{w}}_{r,k}$. Have all $\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k}$ and $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k}$ be outputs of $\tilde{C}$.

(g) Set $\phi(\mathsf{w}) := \tilde{\mathsf{w}}^{\mathsf{out}}_{i,n}$.

3. Set $\vec{\phi}[i] := \mathsf{idx}(\phi(\mathsf{w}))$.

We claim that $\tilde{C}$, produced by the process above, is in the class $\mathbf{C}^{\mathsf{E}}$. Because $C \in \mathbf{C}^{\mathsf{S}}$, it can be seen that all constant and addition gates of $\tilde{C}$ satisfy the conditions of $\mathbf{C}^{\mathsf{E}}$ (see Section 3.3.4 for precise definitions). The same is true for multiplication gates: by induction, every multiplication gate g has its right input $\mathsf{R\text{-}input}(g)$ depend on circuit inputs from a single slot. In particular, the right inputs of $\tilde{g}_{i,k}$, $\tilde{g}^{\mathsf{aux}}_{i,k}$ and $\tilde{g}^{\mathsf{out}}_{i,k}$ discussed above only depend on circuit inputs from slot k.

Finally, wires that are not outputs of $\tilde{C}$ are used in computations in a verifiable way. More precisely, only output gates $\tilde{g}^{\mathsf{aux}}_{i,k}$ and $\tilde{g}^{\mathsf{out}}_{i,k}$ can reference internal wires $\tilde{\mathsf{w}}_{i,k}$ $(k = 2, \ldots, n)$. Therefore we set $\mathsf{mul\text{-}wit}_1(\tilde{C}, \tilde{g}^{\mathsf{aux}}_{i,k}) := \tilde{\mathsf{w}}^{\mathsf{out}}_{i,k-1}$, $\mathsf{mul\text{-}wit}_2(\tilde{C}, \tilde{g}^{\mathsf{aux}}_{i,k}) := \tilde{\mathsf{w}}^{\mathsf{out}}_{l,k}$, $\mathsf{mul\text{-}wit}_3(\tilde{C}, \tilde{g}^{\mathsf{aux}}_{i,k}) := \tilde{\mathsf{w}}^{\mathsf{out}}_{l,k-1}$. If $\mathsf{w}^{\mathsf{R}} \notin \mathsf{inputs}(C)$, we also set $\mathsf{mul\text{-}wit}_1(\tilde{C}, \tilde{g}^{\mathsf{out}}_{i,k}) := \tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k}$, $\mathsf{mul\text{-}wit}_2(\tilde{C}, \tilde{g}^{\mathsf{out}}_{i,k}) := \tilde{\mathsf{w}}^{\mathsf{out}}_{r,k}$, and $\mathsf{mul\text{-}wit}_3(\tilde{C}, \tilde{g}^{\mathsf{out}}_{i,k}) := \tilde{\mathsf{w}}^{\mathsf{out}}_{r,k-1}$. Having made this choice, each gate g that references an internal wire also satisfies

$$\begin{aligned}\mathsf{wire\text{-}poly}(\mathsf{output}(g)) \cdot \mathsf{wire\text{-}poly}(\mathsf{mul\text{-}wit}_3(\tilde{C}, g)) &= \\ \mathsf{wire\text{-}poly}(\mathsf{mul\text{-}wit}_1(\tilde{C}, g)) \cdot \mathsf{wire\text{-}poly}(\mathsf{mul\text{-}wit}_2(\tilde{C}, g)) &,\end{aligned}$$

as required.

We also claim that our $\tilde{C}$ achieves correctness. That is, under multiplicative sharing of inputs, circuit $\tilde{C}$ computes all outputs of C.

Note that, above, for each gate of C, we add $O(n)$ gates and $O(n)$ wires to $\tilde{C}$; hence, $\mathsf{size}(\tilde{C}) = O(n \cdot \mathsf{size}(C))$. Moreover, the gates of $\tilde{C}$ that reference $\tilde{\mathsf{w}}_{\mathsf{in},}$ correspond to gates of C that reference $\mathsf{w}_{\mathsf{in},}$; if a gate g of C references an input wire $\mathsf{w}_{\mathsf{in},}$, then the corresponding $O(n)$ gates of $\tilde{C}$ will reference a wire from each parties' shares $O(1)$ times; hence, $\mathsf{size}(\tilde{C}) = O(n \cdot \mathsf{size}(C))$ and $\mathsf{size}(\tilde{C}, i) = O(\mathsf{size}(C))$ for $i = 1, \ldots, n$.

Finally, one can check that $\mathsf{depth}_{\mathsf{E}}(\tilde{C}) = n \cdot \mathsf{depth}_{\mathsf{S}}(C)$.

Construction of T_2. On input a secure evaluation broadcast protocol $\Pi^{\mathsf{E}} = (\Pi, V, S)$ with n parties for $\tilde{C}$, the transformation T_2 outputs a triple $\Pi^{\mathsf{S}} = (\Pi', V', S')$ that is constructed as follows. (Allegedly, Π^{S} is a secure sampling broadcast protocol with n parties for C.)

- *Construction of* Π'. Let $\Pi = (S, \Sigma_1, \ldots, \Sigma_n)$. Construct $\Pi' := (S', \Sigma'_1, \ldots, \Sigma'_n)$ as follows. The schedule S' is

$$S'(t) := \begin{cases} S(t) & \text{if } 0 < t \leq \mathsf{ROUND}(\Pi) \\ \{1\} & \text{if } t = \mathsf{ROUND}(\Pi) + 1 \\ \emptyset & \text{otherwise} \end{cases} .$$

  Next, for $i = 1, \ldots, n$, the strategy Σ'_i, on input (x_i, t) and with oracle access to the history of messages broadcast so far, works as follows.

  - If $0 < t \leq \mathsf{ROUND}(\Pi)$, run Σ_i on input (x_i, t) and output its output message $\mathsf{msg}_{t,i}$.
  - If $t = \mathsf{ROUND}(\Pi) + 1$ and $i = 1$, do the following. Collect the encoding of the value of every output wire of C. This can be done by using the wire embedding ϕ to select values from the last message broadcast so far because, by definition (being the last message broadcast in Π), this message contains the encoding of the value of every output wire of $\tilde{C}$. Set $\mathsf{msg}_{t,i}$ equal to the vector of these selected encodings and output $\mathsf{msg}_{t,i}$.

  Namely, the first $\mathsf{ROUND}(\Pi)$ rounds of Π' coincide with the first $\mathsf{ROUND}(\Pi)$ rounds of Π. Then, in the last round of Π', party 1 (chosen arbitrarily) collects from the output of Π the encodings needed to create the output for Π'.

- *Construction of* V'. On input a transcript tr, the verifier V' works as follows. Let $\tilde{\mathsf{msg}}$ denote the last message in tr, and $\tilde{\mathsf{tr}}$ the transcript obtained by removing $\tilde{\mathsf{msg}}$ from tr. Check that $V(\tilde{\mathsf{tr}}) = 1$. Then check that $\tilde{\mathsf{msg}}$ equals the message obtained by using the wire embedding ϕ to collect the outputs of C from $\tilde{\mathsf{tr}}$'s last message.

  Finally, to ensure that the adversary cannot bias the output of the sampling protocol by an input value of zero, we perform the following check: whenever an input wire w is used in a multiplication gate g, we check that either g's left input is zero, or g's output is non-zero.

- *Construction of* S'. On input an adversary A and set J of corrupted parties, the simulator S' works as follows. (We assume that $|J| > 0$, for otherwise the simulation is trivial.)

  1. *Construct a new adversary* $\tilde{A}$. The simulator first modifies the adversary A, which is an adversary against the sampling protocol $\Pi^{\mathsf{S}} = (\Pi', V', S')$, to an adversary $\tilde{A}$ against the evaluation protocol $\Pi^{\mathsf{E}} = (\Pi, V, S)$. By construction of Π', this can be done by designing $\tilde{A}$ so that (i) $\tilde{A}$ runs A and lets it interact with the outside world up to and including round $\mathsf{ROUND}(\Pi)$ (up to this round, Π' and Π are identical); and (ii) $\tilde{A}$

simulates for A the last round (the only round at which Π' and Π differ). The last round can be easily simulated by $\tilde{A}$ because, consisting merely of collecting some values from past messages, it is a public operation on the view of A.

2. *Run Π^{E}'s simulator on the new adversary $\tilde{A}$.* The simulator runs S on input the new adversary $\tilde{A}$ and the set J of corrupted parties. When S outputs, for each $i \in J$, the (extracted) malicious input $\vec{\sigma_i}^* := (\sigma_{i,j}^*)_{j=1}^{m_i}$ for party i, the simulator forwards it to the trusted party.

   At the same time, each honest party $i \notin J$ sends to the trusted party his own vector $\sigma_i := (\sigma_{i,j})_{j=1}^{m_i}$. The trusted party, broadcasts the output $f_{n,C,\mathcal{G}}^{\mathsf{S}}(\vec{\sigma}^*)$, where $\vec{\sigma}^*$ combines $(\vec{\sigma_i}^*)_{i \in J}$ and $(\vec{\sigma_i})_{i \notin J}$ in order of i. Note that each malicious input $\vec{\sigma_i}^*$ was not intended as an input to the function $f_{n,C,\mathcal{G}}^{\mathsf{S}}$, but instead to $f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}$. Though, while different, the two functions have the same domain, and thus the trusted party's output is well-defined.

   Next, the simulator must relay to S a value for $f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}$, while only having access to the trusted party's output, $f_{n,C,\mathcal{G}}^{\mathsf{S}}(\vec{\sigma}^*) := C((\prod_{i=1}^{n} \sigma_{i,1}^*, \ldots, \prod_{i=1}^{n} \sigma_{i,m})) \cdot \mathcal{G}$, and the (extracted) malicious inputs, $(\vec{\sigma_i}^*)_{i \in J}$. Crucially, the relayed value must be indistinguishable from the value that S would have seen if S had accessed the function $f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}$. This particular simulation is the core of the simulator and is discussed separately in the next step.

3. *Simulation of $f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}(\vec{\sigma}^*)$.* Let $\vec{\mathcal{X}}$ be the distribution over $\mathbb{F}^{m_1} \times \cdots \times \mathbb{F}^{m_n}$ defined as follows: for each $i \in J$, set $\vec{\chi}_i := \vec{\sigma_i}^*$ (where $\vec{\sigma_i}^*$ is the extracted malicious input for party i); for each $i \notin J$, set $\vec{\chi}_i$ to be random in $(\mathbb{F}^*)^{m_i}$; output $\vec{\chi} := (\vec{\chi}_i)_{i=1}^{n}$. The simulator will relay to S a sample from the distribution

$$\mathcal{D} := \left\{ f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}(\vec{\chi}) \,\middle|\, \vec{\chi} \leftarrow \vec{\mathcal{X}} \right\}_{f_{n,C,\mathcal{G}}^{\mathsf{S}}(\vec{\chi}) = f_{n,C,\mathcal{G}}^{\mathsf{S}}(\vec{\sigma}^*)} .$$

We now explain how the simulator can efficiently generate a sample from $\mathcal{D}$, despite the fact that the simulator does not know the honest parties' inputs (i.e., the i-th coordinate of $\vec{\sigma}^*$ for $i \notin J$).

By construction of $\tilde{C}$, $f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}(\vec{\alpha})$ contains $f_{n,C,\mathcal{G}}^{\mathsf{S}}(\vec{\alpha})$ for any input $\vec{\alpha}$. However, $f_{\tilde{C},\mathcal{G}}^{\mathsf{E}}(\vec{\alpha})$ also contains additional outputs; these are the values of wires that carry partial shares of an output of C. Our strategy is to "compute backwards" all the output wires of $\tilde{C}$, starting from its output wires that are also outputs of C (because these values are the ones we know); this strategy leverages the specific structure of the circuit $\tilde{C}$ and does not apply to every circuit in $\mathbf{C}^{\mathsf{E}}$. More precisely, we proceed as follows.

Arbitrarily choose an honest party $q \in \{1,\ldots,n\} \setminus J$ and let $H := \{1,\ldots,n\} \setminus (J \cup \{q\})$. We will assign random inputs to each party $i \in H$; this, together with (extracted) malicious input $\vec{\sigma}_i^*$, fixes inputs of $n-1$ parties. Note that, together with trusted party's output $f^{\mathsf{S}}_{n,C,\mathcal{G}}(\vec{\sigma}^*)$, the values for parties in $J \cup H$ uniquely determine the inputs of party q and, in turn, uniquely determines $f^{\mathsf{E}}_{\tilde{C},\mathcal{G}}$. Our goal is to compute the encoded outputs $f^{\mathsf{E}}_{\tilde{C},\mathcal{G}}$ without having access to inputs of party q. We do so by using $f^{\mathsf{S}}_{n,C,\mathcal{G}}(\vec{\sigma}^*)$ and inputs of parties in $J \cup H$ to back-compute $f^{\mathsf{E}}_{\tilde{C},\mathcal{G}}$, as described below.

Let $\vec{B}$ be a vector of $\#\mathsf{wires}(\tilde{C})$ coordinates. Letting $m := \#\mathsf{inputs}(C)$ the values of $\vec{B}$ are initialized as follows. For $i \in J$ and $j = 1,\ldots,m$, set $\vec{B}[(m-1)\cdot i + j] := \sigma^*_{i,j}$; for $i \in H$ and $j = 1,\ldots,m$, set $\vec{B}[(m-1)\cdot i + j]$ to be an element drawn uniformly at random from $\mathbb{F}_r^*$; set all other entries of $\vec{B}$ to $\perp$. Intuitively, $\vec{B}$ contains the (plain) wire values of $\tilde{C}$, or $\perp$, if the value is not known. Similarly, let $\vec{E}$ be a vector of $\#\mathsf{wires}(\tilde{C})$ coordinates, initialized as follows. Consult the trusted party's output $f^{\mathsf{S}}_{n,C,\mathcal{G}}(\vec{\sigma}^*)$ and for $k = 1,\ldots,\#\mathsf{outputs}(C)$ set $\vec{E}[\phi(k)] := f^{\mathsf{S}}_{n,C,\mathcal{G}}(\vec{\sigma}^*)[k]$; set all other entries of $\vec{E}$ to $\perp$. Intuitively the k-th coordinate of $\vec{E}$ contains the encoding of the value of the k-th wire in $\tilde{C}$, or $\perp$ is the value is not known.

First, we process the output of constant 1 gate $\tilde{g}_0$ in $\tilde{C}$ by setting $\vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_0)] := 1$ and $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}_0)] := 1 \cdot \mathcal{G}$. All other gates of $\tilde{C}$ arise from a gadget reduction of gates in C, so we process all other gates of $\tilde{C}$ by individually handling each subcircuit of $\tilde{C}$. That is, for each gate g of C, in topological order, we process the corresponding gates and wires in $\tilde{C}$ as follows:

(a) Let $\mathsf{w} := \mathsf{output}(g)$ be the output of g and let $i := \mathsf{idx}(\mathsf{w})$.

(b) Use calculated plain and encoded wire values in $\vec{B}$ and $\vec{E}$ to update the encoded and plain wire values (if applicable) in the subcircuit of $\tilde{C}$ that corresponds to g. We do so by referring to the different cases spelled out in the construction of $\tilde{C}$:

- If g is a constant gate computing $\mathsf{w} \leftarrow \alpha$, set $\vec{B}[\phi(i)] := \alpha$ and $\vec{E}[\phi(i)] := \alpha \cdot \mathcal{G}$.
- If g is an addition gate computing $\mathsf{w} \leftarrow \sum_{j=1}^{d} \alpha_j \mathsf{w}_j$, set $\vec{E}[\phi(i)] := \sum_{j=1}^{d} \alpha_j \cdot \vec{E}[\phi(\mathsf{idx}(\mathsf{w}_j))]$.
- If g is a multiplication gate computing $\mathsf{w} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$, proceed as follows:
  i. We perform all operations using the same notation as described in construction of T_1. That is, let $l := \mathsf{idx}(\mathsf{w}^{\mathsf{L}})$ and $r := \mathsf{idx}(\mathsf{w}^{\mathsf{R}})$. Let $\tilde{g}_{i,1},\ldots,\tilde{g}_{i,n}$; $\tilde{g}^{\mathsf{out}}_{i,2},\ldots,\tilde{g}^{\mathsf{out}}_{i,n}$ and $\tilde{g}^{\mathsf{aux}}_{i,2},\ldots,\tilde{g}^{\mathsf{aux}}_{i,n}$ be the gates in $\tilde{C}$ added for handling g. Finally, let $\tilde{\mathsf{w}}_{i,1},\ldots,\tilde{\mathsf{w}}_{i,n}$; $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,2},\ldots,\tilde{\mathsf{w}}^{\mathsf{out}}_{i,n}$ and $\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,2},\ldots,\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,n}$ be their respective outputs.

ii. Note that for $2 \leq k \leq n$, $k \neq q$ the simulator has filled in the values of $\vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{l,k})]$ and $\vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{r,k})]$ (in particular, the simulator has done so for every multiplication gate; for addition and constant gates the wires $\tilde{\mathsf{w}}_{l,k}$ and $\tilde{\mathsf{w}}_{r,k}$ refer to $\tilde{\mathsf{w}}_0$). So for all such k, the simulator sets $\vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{i,k})] := \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{l,k})] \cdot \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{r,k})]$. The encoded values of $\tilde{\mathsf{w}}_{i,k}$'s are not circuit outputs and, thus, are not output by the ideal functionality, so we don't compute them.

iii. Next, simulator computes the encoded values of $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k}$'s and $\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k}$'s in two passes. First, it starts by encoding of $\tilde{\mathsf{w}}_{l,1}$ and uses the plain values of $\tilde{\mathsf{w}}_{r,1}, \tilde{\mathsf{w}}_{l,2}, \tilde{\mathsf{w}}_{r,2}, \ldots, \tilde{\mathsf{w}}_{r,q-1}$ to compute in forward direction. Next, it uses the plain values of $\tilde{\mathsf{w}}_{r,n}, \tilde{\mathsf{w}}_{l,n}, \tilde{\mathsf{w}}_{r,n-1}, \ldots, \tilde{\mathsf{w}}_{l,q+1}$ to compute "backwards" from the encoding of $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,n}$. Note that the simulator knows the encoding of $\tilde{\mathsf{w}}^{\mathsf{out}}_{i,n}$ as this value is part of the trusted party's output.

More precisely, if $q > 1$, the simulator computes $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{i,1})] := \vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{l,1})] \cdot \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{r,1})]$ and for $k = 2, \ldots, q-1$ computes $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k})] := \vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k-1})] \cdot \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{l,k})]$ and $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k})] := \vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k})] \cdot \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{r,k})]$. Similarly for $k = n, \ldots, q+1$, the simulator computes $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k})] := \vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k})] / \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{r,k})]$ and $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{i,k-1})] := \vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,k})] / \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{l,k})]$.

iv. If $q = 1$, the simulator has handled all output wires of $\tilde{C}$ that correspond to g in C. If $q > 1$, the simulator has handled all wires except $\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,q}$, so it sets $\vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{aux}}_{i,q})] := \vec{E}[\mathsf{idx}(\tilde{\mathsf{w}}^{\mathsf{out}}_{l,1})] \cdot \left(\prod_{k=1}^{q-1} \vec{B}[\mathsf{idx}(\tilde{\mathsf{w}}_{r,k})]\right) \cdot \alpha$.

4. *Extend the output of S.* Extend $\tilde{\mathsf{tr}}$ with an additional message, $f^{\mathsf{S}}_{n,C,\mathcal{G}}(\vec{\sigma}^*)$, and denote the result by tr. This last message reflects the additional round present in Π' (as compared to Π), and its goal is merely to re-format the output of the protocol. Also, since extending $\tilde{\mathsf{tr}}$ to tr does not require additional randomness, we can set $r := \tilde{r}$.

5. *Output.* Output tr (the transcript), $(\vec{\sigma}_i)_{i \in J}$ (the inputs of the corrupted parties), and r (the adversary's randomness).

3.11 Proof of Lemma 3.5.3

We prove Lemma 3.5.3. Specifically, first we describe the construction of $\Pi^{\mathsf{E}} = (\Pi, V, S)$ by describing, for every positive integer n and circuit $C \colon \mathbb{F}_r^{m_1} \times \cdots \times \mathbb{F}_r^{m_n} \to \mathbb{F}_r^h$ in $\mathbf{C}^{\mathsf{E}}$, the multi-party broadcast protocol $\Pi_{n,C}$, the verifier $V_{n,C}$, and the simulator $S_{n,C}$; afterwards, we explain why the construction of Π^{E} works.

Below, we use the following cryptographic ingredients: a commitment scheme COMM (see Section 3.3.2), and three NIZKs (see Section 3.3.3), namely, $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}$ for the NP relation $\mathcal{R}_{\mathrm{A}}$, $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{B,inp}}}$ for $\mathcal{R}_{\mathrm{B,inp}}$, and $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{B,priv}}}$ for $\mathcal{R}_{\mathrm{B,priv}}$; these three relations are defined in Figure 3.4. Parties have access to a common random string crs that we parse as $(\mathsf{crs}_\star, \mathsf{crs}_{\mathcal{R}_{\mathrm{A}}}, \mathsf{crs}_{\mathcal{R}_{\mathrm{B,inp}}}, \mathsf{crs}_{\mathcal{R}_{\mathrm{B,inp}}})$ where $\mathsf{crs}_\star$ is a common random string for COMM, $\mathsf{crs}_{\mathcal{R}_{\mathrm{A}}}$ for $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}$, $\mathsf{crs}_{\mathcal{R}_{\mathrm{B,inp}}}$ for $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{B,inp}}}$ and $\mathsf{crs}_{\mathcal{R}_{\mathrm{B,priv}}}$ for $\mathsf{NIZK}_{\mathcal{R}_{\mathrm{B,priv}}}$.

Construction of $\Pi_{n,C}$. The n-party broadcast protocol $\Pi_{n,C}$ is a tuple $(S, \Sigma_1, \ldots, \Sigma_n)$ that is constructed as follows. The schedule S is

$$S(t) := \begin{cases} \{1, \ldots, n\} & \text{if } t = 1 \\ \{i \mid \exists\, \mathsf{w} \in \mathsf{outputs}(C) \text{ s.t. } \mathsf{depth}_{\mathsf{E}}(\mathsf{w}) = t - 1 \text{ and } i \in \mathsf{ds}(\mathsf{w})\} & \text{if } 1 < t \leq \mathsf{depth}_{\mathsf{E}}(C) + 1 \\ \{1\} & \text{if } t = \mathsf{depth}_{\mathsf{E}}(C) + 2 \\ \emptyset & \text{otherwise} \end{cases} .$$

Next, for $i = 1, \ldots, n$, the strategy Σ_i, on input (x_i, t) and with oracle access to the history of messages broadcast so far, works as follows.

- If $t = 1$, do the following.
  1. Parse $\mathsf{inputs}(C, i)$ as $\{\mathsf{w}_{i,j}\}_{j=1}^{m_i}$ and x_i as $(\sigma_{i,j})_{j=1}^{m_i}$; each $\sigma_{i,j}$ lies in $\mathbb{F}_r$ and represents the value of wire $\mathsf{w}_{i,j}$.
  2. Set $U_i := \left((\mathsf{w}_{i,j}, \sigma_{i,j})\right)_{j=1}^{m_i}$. The protocol will maintain the invariant that U_i contains all private values known to the party i.
  3. For $j = 1, \ldots, m_i$:
     (a) sample the commitment and randomness $(\mathsf{cm}_{i,j}, \mathsf{cr}_{i,j}) \leftarrow \mathsf{COMM}.\mathsf{G}(\mathsf{crs}_\star, \sigma_{i,j})$;
     (b) construct the instance $\mathbb{x}_{i,j} := (\mathsf{crs}_\star, \mathsf{cm}_{i,j})$ and witness $\mathbb{w}_{i,j} := (\sigma_{i,j}, \mathsf{cr}_{i,j})$;
     (c) compute the NIZK proof $\pi_{i,j} := \mathsf{NIZK}_{\mathcal{R}_{\mathrm{A}}}.\mathsf{P}(\mathsf{crs}_{\mathcal{R}_{\mathrm{A}}}, \mathbb{x}_{i,j}, \mathbb{w}_{i,j})$.
  4. Store, for later use, the list U_i and commitment randomness $(\mathsf{cr}_{i,1}, \ldots, \mathsf{cr}_{i,m_i})$.
  5. Output the message $\mathsf{msg}_{t,i} := (\mathsf{cm}_{i,1}, \pi_{i,1}, \ldots, \mathsf{cm}_{i,m_i}, \pi_{i,m_i})$.
- If $1 < t \leq \mathsf{depth}_{\mathsf{E}}(C) + 1$, do the following. Define the set of wires

$$W_{t,i} := \left\{ \mathsf{w} \in \mathsf{outputs}(C) \;\middle|\; \begin{array}{c} \mathsf{depth}_{\mathsf{E}}(\mathsf{w}) = t - 1 \,\wedge \\ i \in \mathsf{ds}(\mathsf{w}) \end{array} \right\} . \tag{3.1}$$

If $W_{t,i}$ is empty, return an empty list as the party's output for the round (since party i has no gates to process during this round). Otherwise, initialize the message $\mathsf{msg}_{t,i}$ to be an empty list and, for each wire $\mathsf{w} \in W_{t,i}$ taken in topological order from inputs towards outputs, perform the following steps.

1. Let g be the gate in $\mathsf{gates}(C)$ whose output wire is w.
2. If $\mathsf{type}(g) = \mathsf{const}$:
   - Say that the gate g computes $\mathsf{w} \leftarrow \alpha$ for $\alpha \in \mathbb{F}_r$.
   - Compute the $\mathbb{G}$-element $\mathcal{R} := \alpha \cdot \mathcal{G}$; $\mathcal{R}$ encodes the value of the output wire w.
   - Append (w, α) to U_i.
   - Append $(\mathsf{w}, \mathcal{R}, \perp)$ to $\mathsf{msg}_{t,i}$.
3. If $\mathsf{type}(g) = \mathsf{add}$:
   - Say that the gate g computes $\mathsf{w} \leftarrow \sum_{j=1}^{d} \alpha_j \mathsf{w}_j$ for $\alpha_1, \ldots, \alpha_d \in \mathbb{F}_r$.
   - For $j = 1, \ldots, d$, consult the history of messages broadcast so far (or previous iterations of this loop) to obtain a tuple $(\mathsf{w}_j, \mathcal{P}_j, \pi_j)$; $\mathcal{P}_j$ encodes the value of the j-th input wire w_j and π_j is a NIZK proof attesting to this.
   - Compute the $\mathbb{G}$-element $\mathcal{R} := \sum_{j=1}^{d} (\alpha_j \cdot \mathcal{P}_j)$; $\mathcal{R}$ encodes the value of the output wire w.
   - If U_i contains pairs (w_j, σ_j) for all $j = 1, \ldots, d$, then compute $\sigma := \sum_{j=1}^{d} \alpha_j \sigma_j$ and append (w, σ) to U_i.
   - Append $(\mathsf{w}, \mathcal{R}, \perp)$ to $\mathsf{msg}_{t,i}$.
4. If $\mathsf{type}(g) = \mathsf{mul}$:
   - Say that the gate g computes $\mathsf{w} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$ for $\alpha \in \mathbb{F}_r$.
   - Consult the history of messages broadcast so far (or previous iterations of this loop) to obtain a tuple $(\mathsf{w}^{\mathsf{L}}, \mathcal{P}^{\mathsf{L}}, \pi^{\mathsf{L}})$; $\mathcal{P}^{\mathsf{L}}$ encodes the value of the left input wire w^{L} and π^{L} is a NIZK proof attesting to this.
   - Consult U_i to obtain a pair $(\mathsf{w}^{\mathsf{R}}, \sigma^{\mathsf{R}})$; the $\mathbb{F}_r$-element σ^{R} is the value of the right input wire w^{R}. (This can always be done, because $C \in \mathbf{C}^{\mathsf{E}}$ and so w^{R} depends only on inputs of party i.)
   - If U_i contains a pair $(\mathsf{w}^{\mathsf{L}}, \sigma^{\mathsf{L}})$, then compute $\sigma := \alpha \sigma^{\mathsf{L}} \sigma^{\mathsf{R}}$ and append (w, σ) to U_i.
   - Compute the $\mathbb{G}$-element $\mathcal{R} := \alpha \sigma^{\mathsf{R}} \cdot \mathcal{P}^{\mathsf{L}}$; $\mathcal{R}$ encodes the value of the output wire w.
   - If $\mathsf{w}^{\mathsf{R}} \in \mathsf{inputs}(C)$:
     (a) let j' be the index such that w^{R} is the j'-th wire in $\mathsf{inputs}(C, i)$;
     (b) set $\mathsf{cm} := \mathsf{cm}_{i,j'}$, $\mathsf{cr} := \mathsf{cr}_{i,j'}$, $\alpha := \alpha$, $\mathcal{P} := \mathcal{P}^{\mathsf{L}}$;
     (c) construct the instance $\mathbb{x} := (\mathsf{crs}_{\star}, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$ and witness $\mathbb{w} := (\sigma, \mathsf{cr})$; the pair $(\mathbb{x}, \mathbb{w})$ belongs to $\mathcal{R}_{\mathsf{B},\mathsf{inp}}$;

(d) compute the NIZK proof $\pi := \mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{P}(\mathsf{crs}_{\mathcal{R}_{\mathsf{B,inp}}}, \mathbb{x}, \mathbb{w})$.

- If $\mathsf{w}^{\mathsf{R}} \notin \mathsf{inputs}(C)$:
  (a) consult the history of messages broadcast so far (or previous iterations of this loop) to obtain triples $(\mathsf{mul\text{-}wit}_1(C,g), \mathcal{P}, \pi_{\mathcal{P}})$, $(\mathsf{mul\text{-}wit}_2(C,g), \mathcal{Q}_1, \pi_{\mathcal{Q},1})$ and $(\mathsf{mul\text{-}wit}_3(C,g), \mathcal{Q}_2, \pi_{\mathcal{Q},2})$; this can be done because each of the wires $\mathsf{mul\text{-}wit}_1(C,g)$, $\mathsf{mul\text{-}wit}_2(C,g)$ and $\mathsf{mul\text{-}wit}_1(C,g)$ are circuit outputs, due to topological ordering, that were processed before w;

  (b) consult U_i to obtain pairs $(\mathsf{mul\text{-}wit}_2(C,g), \sigma_1)$ and $(\mathsf{mul\text{-}wit}_3(C,g), \sigma_2)$; this can be done because $C \in \mathbf{C}^{\mathsf{E}}$ and so $\mathsf{mul\text{-}wit}_2(C,g)$ and $\mathsf{mul\text{-}wit}_3(C,g)$ depend only on inputs of party i;

  (c) construct the instance $\mathbb{x} := (\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2, \alpha)$ and witness $\mathbb{w} := (\sigma_1, \sigma_2)$; the pair $(\mathbb{x}, \mathbb{w})$ belongs to $\mathcal{R}_{\mathsf{B,priv}}$;

  (d) compute the NIZK proof $\pi := \mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,priv}}}.\mathsf{P}(\mathsf{crs}_{\mathcal{R}_{\mathsf{B,priv}}}, \mathbb{x}, \mathbb{w})$.
- Append $(\mathsf{w}, \mathcal{R}, \pi)$ to $\mathsf{msg}_{t,i}$.

Output the message $\mathsf{msg}_{t,i}$.

- If $t = 2 + \mathsf{depth}_{\mathsf{E}}(C)$ and $i = 1$, do the following.
  1. Parse $\mathsf{outputs}(C)$ as $\{\mathsf{w}_{\mathsf{out},j}\}_{j=1}^{h}$.
  2. Consult the history of messages broadcast so far to collect the encoding of every output of C, i.e., to collect the triples $\big((\mathsf{w}_j, \mathcal{R}_j, \pi_j)\big)_{j=1}^{h}$ with $\mathsf{w}_j = \mathsf{w}_{\mathsf{out},j}$.
  3. Output the message $\mathsf{msg}_{t,i} := (\mathcal{R}_j)_{j=1}^{h}$.

Construction of $V_{n,C}$. On input a transcript tr, the verifier $V_{n,C}$ first uses $\Pi_{n,C}$'s schedule to parse tr as a sequence of messages $\mathsf{msg}_{t,i}$ where $\mathsf{msg}_{t,i}$ is the t-th message broadcast by party i. Here, t ranges from 1 to $2 + \mathsf{depth}_{\mathsf{E}}(C)$ and i ranges from 1 to n; if tr cannot be parsed in this way, $V_{n,C}$ rejects. Next, the verifier performs the following checks.

- *Check that parties have collectively committed to a valid input.* For $i = 1, \dots, n$, check that $\mathsf{msg}_{1,i}$ equals a vector of m_i commitments and NIZK proofs, which we denote by $(\mathsf{cm}_{i,1}, \pi_{i,1}, \dots, \mathsf{cm}_{i,m_i}, \pi_{i,m_i})$, and that, for $j = 1, \dots, m_i$, $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{A}}}.\mathsf{V}(\mathsf{crs}_{\mathcal{R}_{\mathsf{A}}}, (\mathsf{crs}_{\star}, \mathsf{cm}_{i,j}), \pi_{i,j}) = 1$.
- *Check that each gate in C is correctly evaluated.* For $i = 1, \dots, n$ and $t = 2, \dots, 1 + \mathsf{depth}_{\mathsf{E}}(C)$ do the following. First define the set $W_{t,i}$ as in Equation 3.1. Then check that $\mathsf{msg}_{t,i}$ equals a list of $|W_{t,i}|$ triples $(\mathsf{w}_{t,i,j}, \mathcal{R}_{t,i,j}, \pi_{t,i,j})$, where each $\mathsf{w}_{t,i,j}$ is a wire of C, each $\mathcal{R}_{t,i,j}$ is an element of $\mathbb{G}$, and each $\pi_{t,i,j}$ is a NIZK proof or $\perp$; also, check that $\{\mathsf{w}_{t,i,j}\}_j = W_{t,i}$. Finally, check that for each j the encoding of $\mathsf{w}_{t,i,j}$ was correctly computed. There are four cases to consider:

- Case 1: $\mathsf{w}_{t,i,j}$ is an output wire of a constant gate $\mathsf{w}_{t,i,j} \leftarrow \alpha$. Check that $\mathcal{R}_{t,i,j} = \alpha \cdot \mathcal{G}$.
- Case 2: $\mathsf{w}_{t,i,j}$ is an output wire of an addition gate $\mathsf{w}_{t,i,j} \leftarrow \sum_{k=1}^{d} \alpha_k \mathsf{w}_k$. Consult the transcript to obtain triples $(\mathsf{w}_k, \mathcal{P}_k, \pi_k)$ for $k = 1, \dots, d$, and then check that $\mathcal{R}_{t,i,j} = \sum_{k=1}^{d} \alpha_k \cdot \mathcal{P}_k$.
- Case 3: $\mathsf{w}_{t,i,j}$ is an output wire of a multiplication gate $\mathsf{w}_{t,i,j} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$ and $\mathsf{w}^{\mathsf{R}} \in \mathsf{inputs}(C)$. Check that $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{V}(\mathsf{crs}_{\mathcal{R}_{\mathsf{B,inp}}}, \mathbb{x}_{t,i,j}, \pi_{t,i,j}) = 1$, where $\mathbb{x}_{t,i,j} := (\mathsf{crs}_\star, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$ is an instance for the NP relation $\mathcal{R}_{\mathsf{B,inp}}$ that is constructed (analogously to Step 4 above in Σ_i's description), as follows: consult the transcript to obtain a triple $(\mathsf{w}^{\mathsf{L}}, \mathcal{P}^{\mathsf{L}}, \pi^{\mathsf{L}})$ and set $\mathcal{R} := \mathcal{R}_{t,i,j}$, $\mathcal{P} := \mathcal{P}^{\mathsf{L}}$, and $\alpha := \alpha$; then, letting j' be the index such that w^{R} is the j'-th wire in $\mathsf{inputs}(C, i)$, set $\mathsf{cm} := \mathsf{cm}_{i,j'}$.
- Case 4: $\mathsf{w}_{t,i,j}$ is an output wire of a multiplication gate $\mathsf{w}_{t,i,j} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$ and $\mathsf{w}^{\mathsf{R}} \notin \mathsf{inputs}(C)$. Check that $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,priv}}}.\mathsf{V}(\mathsf{crs}_{\mathcal{R}_{\mathsf{B,priv}}}, \mathbb{x}_{t,i,j}, \pi_{t,i,j}) = 1$, where $\mathbb{x}_{t,i,j} := (\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2, \alpha)$ is an instance for the NP relation $\mathcal{R}_{\mathsf{B,priv}}$ that is constructed (analogously to Step 4 above in Σ_i's description), by consulting the transcript to obtain the triples $(\mathsf{mul\text{-}wit}_1(C, \mathsf{w}), \mathcal{P}, \pi_{\mathcal{P}})$, $(\mathsf{mul\text{-}wit}_2(C, \mathsf{w}), \mathcal{Q}_1, \pi_{\mathcal{Q},1})$, and $(\mathsf{mul\text{-}wit}_3(C, g), \mathcal{Q}_2, \pi_{\mathcal{Q},2})$.

  Moreover, check that $\mathcal{Q}_2$ does not equal the identity element of the group $\mathbb{G}$; this check ensures that the value of the wire $\mathsf{mul\text{-}wit}_3(C, g)$ encoded in $\mathcal{Q}_2$ does not equal zero. Taken together, all such checks ensure that C is evaluated on a some valid input; and, conversely, an honest protocol execution on any valid input will pass these tests.

- *Check that party* 1 *collected all the encodings of outputs.* Collect, among the aforementioned triples of the form $(\mathsf{w}, \mathcal{R}, \pi)$, encodings of the values of output wires of C, and check that the vector whose entries equals these encodings matches the message $\mathsf{msg}_{2+\mathsf{depth}_{\mathsf{E}}(C),1}$.

Construction of $S_{n,C}$. On input an adversary A and set J of corrupted parties, the simulator $S_{n,C}$ works as follows.

1. *Initialization.* The simulator initializes an empty transcript tr; over the course of running the adversary A with randomness r, the simulator will add to tr both simulated messages (on behalf of honest parties) and messages output by A (on behalf of corrupted parties). The simulator samples a common random string $\mathsf{crs}_\star$ for COMM and common random strings for the three NIZKs, with an extraction trapdoor for $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{A}}}$ and simulation trapdoors for $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}$ and $\mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,priv}}}$, i.e., $(\mathsf{crs}^{\mathsf{ext}}_{\mathcal{R}_{\mathsf{A}}}, \mathsf{trap}^{\mathsf{ext}}_{\mathcal{R}_{\mathsf{A}}}) \leftarrow \mathsf{NIZK}_{\mathcal{R}_{\mathsf{A}}}.\mathsf{E}_1$, $(\mathsf{crs}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,inp}}}, \mathsf{trap}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,inp}}}) \leftarrow \mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{S}_1$ and $(\mathsf{crs}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,priv}}}, \mathsf{trap}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,priv}}}) \leftarrow \mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,priv}}}.\mathsf{S}_1$. The common random string shown to A is $\mathsf{crs} := (\mathsf{crs}_\star, \mathsf{crs}^{\mathsf{ext}}_{\mathcal{R}_{\mathsf{A}}}, \mathsf{crs}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,inp}}}, \mathsf{crs}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,priv}}})$. The simulator samples a random string r for the adversary A, and then runs A, on inputs $(\vec{\sigma}_i)_{i \in J}$ and with randomness r, until A asks for the first round's messages of the honest parties.

2. *Simulation of the first round.* The adversary A expects, for each honest party $i \notin J$, a message $\mathsf{msg}_{1,i}$. The simulator, for each honest party $i \notin J$, does the following. For $j = 1, \dots, m_i$, pick the dummy value $\rho_{i,j} = 0$ and sample commitment and randomness $(\mathsf{cm}_{i,j}, \mathsf{cr}_{i,j}) \leftarrow \mathsf{COMM.G}(\mathsf{crs}_\star, \rho_{i,j})$, construct the instance $\mathbb{x}_{i,j} := (\mathsf{crs}_\star, \mathsf{cm}_{i,j})$ and witness $\mathbb{w}_{i,j} := (\rho_{i,j}, \mathsf{cr}_{i,j})$, and compute the NIZK proof $\pi_{i,j} := \mathsf{NIZK}_{\mathcal{R}_\mathrm{A}}.\mathsf{P}(\mathsf{crs}^{\mathsf{ext}}_{\mathcal{R}_\mathrm{A}}, \mathbb{x}_{i,j}, \mathbb{w}_{i,j})$; then answer with the message $\mathsf{msg}_{1,i} := (\mathsf{cm}_{i,1}, \pi_{i,1}, \dots, \mathsf{cm}_{i,m_i}, \pi_{i,m_i})$.

   The adversary outputs, for each $i \in J$, a message $\mathsf{msg}_{1,i}$ of his choice. The simulator adds all these messages (the messages that are simulated and those that are output by the adversary) to the transcript tr.

3. *Invocation of the trusted party.* The simulator, for each corrupted party $i \in J$, does the following. For $j = 1, \dots, m_i$, extract $\sigma^*_{i,j}$, the j-th input chosen by the adversary for party i, from the commitment $\mathsf{cm}_{i,j}$ and proof $\pi_{i,j}$ in $\mathsf{msg}_{1,i}$, by computing $\sigma^*_{i,j} := \mathsf{NIZK}_{\mathcal{R}_\mathrm{A}}.\mathsf{E}_2(\mathsf{crs}^{\mathsf{ext}}_{\mathcal{R}_\mathrm{A}}, \mathsf{trap}^{\mathsf{ext}}_{\mathcal{R}_\mathrm{A}}, \mathsf{cm}_{i,j}, \pi_{i,j})$; then send to the trusted party the vector $\vec{\sigma_i}^* := (\sigma^*_{i,j})_{j=1}^{m_i}$ as the private input of party i.

   At the same time, each honest party $i \notin J$ sends to the trusted party his own vector $\sigma_i := (\sigma_{i,j})_{j=1}^{m_i}$. The trusted party, broadcasts the output $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}^*)$ where $\vec{\sigma}^*$ combines $(\vec{\sigma_i}^*)_{i \in J}$ and $(\vec{\sigma_i})_{i \notin J}$ in order of i. If $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}^*) = \bot$, then halt and output the special symbol abort (this corresponds to the case $\vec{\sigma}^* \notin \mathsf{valid\text{-}inputs}(C)$); else it is the case that $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}^*) := C(\vec{\sigma}^*) \cdot \mathcal{G}$, and the simulator continues as below.

   If at any point of the execution the simulator can't perform an operation, it halts and outputs the special symbol abort. This represents the case where, in the real world, the adversary produces syntactically invalid messages.

4. *Parsing the trusted party's output.* The simulator re-organizes $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}^*)$ into a data structure that allows for easy lookup of information in the next step: initialize E to be an empty list; then, for each output wire w of C, add to E the pair $(\mathsf{w}, \mathcal{R})$ where $\mathcal{R}$ encodes w's value (and can be found in $f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}^*)$ by definition).

5. *Simulation of the rounds* 2 *through* $1 + \mathsf{depth}_\mathsf{E}(C)$. For t ranging from 2 to $1 + \mathsf{depth}_\mathsf{E}(C)$, the adversary expects, for each honest party $i \notin J$, a message $\mathsf{msg}_{t,i}$. The simulator, for each honest party $i \notin J$, does the following. Define $W_{t,i}$ as in Equation 3.1. Initialize $\mathsf{msg}_{t,i}$ as an empty list and, for each output wire $\mathsf{w} \in W_{t,i}$ taken in topological order, perform the following steps, depending on whether g is a constant gate, addition gate, or multiplication gate.

- If $\mathsf{type}(g) = \mathsf{const}$:
  (a) Say that the gate g computes $\mathsf{w} \leftarrow \alpha$ for $\alpha \in \mathbb{F}_r$.
  (b) Compute the $\mathbb{G}$-element $\mathcal{R} := \alpha \cdot \mathcal{G}$.
  (c) Append $(\mathsf{w}, \mathcal{R}, \perp)$ to $\mathsf{msg}_{t,i}$.
- If $\mathsf{type}(g) = \mathsf{add}$:
  (a) Say that the gate g computes $\mathsf{w} \leftarrow \sum_{j=1}^{d} \alpha_j \mathsf{w}_j$ for $\alpha_1, \ldots, \alpha_d \in \mathbb{F}_r$.
  (b) For $j = 1, \ldots, d$, consult E to obtain the $\mathbb{G}$-element $\mathcal{R}_j$ that encodes w_j's value. Note that $C \in \mathbf{C}^{\mathsf{E}}$ implies that each w_j is in $\mathsf{outputs}(C)$ so that its value can be found in E.
  (c) Compute the $\mathbb{G}$-element $\mathcal{R} := \sum_{j=1}^{d} \alpha_j \mathcal{R}_j$.
  (d) Append $(\mathsf{w}, \mathcal{R}, \perp)$ to $\mathsf{msg}_{t,i}$.
- If $\mathsf{type}(g) = \mathsf{mul}$:
  (a) Say that the gate g computes $\mathsf{w} \leftarrow \alpha \mathsf{w}^{\mathsf{L}} \mathsf{w}^{\mathsf{R}}$ for $\alpha \in \mathbb{F}_r$.
  (b) If $\mathsf{w}^{\mathsf{R}} \in \mathsf{inputs}(C)$:
    - Consult E to obtain the $\mathbb{G}$-element $\mathcal{P}^{\mathsf{L}}$ that encodes w^{L}'s value. Note that $C \in \mathbf{C}^{\mathsf{E}}$ implies that w^{L} is a circuit output, thus its value can be found in E.
    - Consult E to obtain the $\mathbb{G}$-element $\mathcal{R}$ that encodes w's value.
    - Letting j' be the index such that w^{R} is the j'-th wire in $\mathsf{inputs}(C, i)$, set $\mathsf{cm} := \mathsf{cm}_{i,j'}$ (where $\mathsf{cm}_{i,j'}$ is in $\mathsf{msg}_{1,i}$), $\alpha := \alpha$ and $\mathcal{P} := \mathcal{P}^{\mathsf{L}}$.
    - Construct the instance $\mathbb{x} := (\mathsf{crs}, \mathsf{cm}, \mathcal{R}, \mathcal{P}, \alpha)$.
    - Compute the NIZK proof $\pi := \mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,inp}}}.\mathsf{S}_2(\mathsf{crs}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,inp}}}, \mathsf{trap}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,inp}}}, \mathbb{x})$.
  (c) If $\mathsf{w}^{\mathsf{R}} \notin \mathsf{inputs}(C)$:
    - Consult E to obtain the $\mathbb{G}$-elements $\mathcal{P}$, $\mathcal{Q}_1$, $\mathcal{Q}_2$ that encode the values of $\mathsf{mul\text{-}wit}_1(C, g)$, $\mathsf{mul\text{-}wit}_2(C, g)$ and $\mathsf{mul\text{-}wit}_3(C, g)$, respectively. This can be done because each of the wires $\mathsf{mul\text{-}wit}_1(C, g)$, $\mathsf{mul\text{-}wit}_2(C, g)$ and $\mathsf{mul\text{-}wit}_1(C, g)$ are circuit outputs, due to topological ordering, were processed before w.
    - Consult E to obtain the $\mathbb{G}$-element $\mathcal{R}$ that encodes w's value.
    - Construct the instance $\mathbb{x} := (\mathcal{R}, \mathcal{P}, \mathcal{Q}_1, \mathcal{Q}_2, \alpha)$.
    - Compute the NIZK proof $\pi := \mathsf{NIZK}_{\mathcal{R}_{\mathsf{B,priv}}}.\mathsf{S}_2(\mathsf{crs}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,priv}}}, \mathsf{trap}^{\mathsf{sim}}_{\mathcal{R}_{\mathsf{B,priv}}}, \mathbb{x})$.
  (d) Append $(\mathsf{w}, \mathcal{R}, \pi)$ to $\mathsf{msg}_{t,i}$.

Answer the adversary with the message $\mathsf{msg}_{t,i}$.

The adversary outputs, for each $i \in J$, a message $\mathsf{msg}_{t,i}$ of his choice. The simulator adds all the round-t messages (the messages that are simulated and those that are output by the adversary) to the transcript tr.

6. *Simulation of the last round.* We distinguish between two cases:

   - If $1 \notin J$, the adversary expects a message $\mathsf{msg}_{2+\mathsf{depth}_{\mathsf{E}}(C),1}$; the simulator provides $\mathsf{msg}_{2+\mathsf{depth}_{\mathsf{E}}(C),1} := f^{\mathsf{E}}_{C,\mathcal{G}}(\vec{\sigma}^*)$.
   - If $1 \in J$, the adversary does not expect any messages and instead outputs a message $\mathsf{msg}_{2+\mathsf{depth}_{\mathsf{E}}(C),1}$.

   In either case, the simulator adds the last-round message $\mathsf{msg}_{2+\mathsf{depth}_{\mathsf{E}}(C),1}$ to the transcript tr.

7. *Verification of the transcript.* Check that $V(\mathrm{tr}) = 1$; if not, then halt and output the special symbol abort. This represents the case where, in the real world, the adversary has produced syntactically valid messages that, however, do not pass the semantic checks, e.g. invalid NIZK proofs, etc. Of course, all messages produced by the simulator on the behalf of the honest parties are syntactically and semantically valid and do not by themselves make V reject.

8. *Output.* Output tr (the transcript), $(\vec{\sigma}_i)_{i \in J}$ (the inputs of the corrupted parties), and r (the adversary's randomness).

3.12 Examples of circuits underlying generators

As discussed in Section 3.1.2, the generator G of essentially all known (preprocessing) zk-SNARK constructions follows the same computation pattern. To generate the public parameters pp for a given NP relation $\mathcal{R}$, G first constructs an $\mathbb{F}_r$-arithmetic circuit $C \colon \mathbb{F}_r^m \to \mathbb{F}_r^h$ (which is somehow related to $\mathcal{R}$), then samples $\vec{\alpha}$ in $\mathbb{F}_r^m$ at random, and finally outputs $\mathsf{pp} := C(\vec{\alpha}) \cdot \mathcal{G}$ (where $\mathcal{G}$ generates a certain group of order r). Different zk-SNARK constructions differ in (i) which NP relations $\mathcal{R}$ are "natively" supported, and (ii) how the circuit C is obtained from $\mathcal{R}$.

Below, we give two examples of how the generator of a known zk-SNARK construction can be cast in the above paradigm and, moreover, the resulting circuit C lies in the class $\mathbf{C}^{\mathsf{S}}$. Throughout, we denote by $\mathbb{F}[z]$ the ring of univariate polynomials over $\mathbb{F}$, and by $\mathbb{F}^{\leq d}[z]$ the subring of polynomials of degree $\leq d$.

3.12.1 Example for a QAP-based zk-SNARK

We describe how to cast the generator of [PGHR13]'s zk-SNARK as computing the encoding of a random evaluation of a circuit C that lies in $\mathbf{C}^{\mathsf{S}}$. More precisely, we consider [BCTV14c]'s zk-SNARK, which modifies [PGHR13]'s.

Supported NP relations. This zk-SNARK supports arithmetic circuit satisfiability (see Footnote 5), i.e., relations of the form $\mathcal{R}_D = \{(\vec{x}, \vec{w}) \in \mathbb{F}_r^n \times \mathbb{F}_r^h : D(\vec{x}, \vec{w}) = 0^\ell\}$ where $D \colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^\ell$ is an $\mathbb{F}_r$-arithmetic circuit.

QAPs. The construction is based on *quadratic arithmetic programs* (QAP) [GGPR13]: a QAP of size m and degree d over $\mathbb{F}$ is a tuple $(\vec{A}, \vec{B}, \vec{C}, Z)$, where $\vec{A}, \vec{B}, \vec{C}$ are three vectors, each of $m + 1$ polynomials in $\mathbb{F}^{\leq d-1}[z]$, and $Z \in \mathbb{F}[z]$ has degree exactly d. As shown in [GGPR13], each relation $\mathcal{R}_D$ can be reduced to a certain relation $\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, which captures "QAP satisfiability", by computing $(\vec{A}, \vec{B}, \vec{C}, Z) := \mathsf{GetQAP}(D)$ for a suitable function GetQAP; if D has N_{w} wires and N_{g} gates, then the resulting QAP has size $m = N_{\mathsf{w}}$ and degree $d \approx N_{\mathsf{g}}$.

The parameter generator. On input an $\mathbb{F}_r$-arithmetic circuit $D \colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^\ell$, the generator does the following.

1. Compute $(\vec{A}, \vec{B}, \vec{C}, Z) := \mathsf{GetQAP}(D)$, and denote by m and d the QAP's size and degree; then construct an $\mathbb{F}_r$-arithmetic circuit $C \colon \mathbb{F}_r^8 \to \mathbb{F}_r^{d+7m+n+22}$ such that $C(\tau, \rho_{\mathsf{A}}, \rho_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \beta, \gamma)$ computes the following outputs:

$$
\begin{aligned}
&\Big(1, \tau, \ldots, \tau^d, \\
&A_0(\tau)\rho_{\mathsf{A}}, \ldots, A_m(\tau)\rho_{\mathsf{A}}, Z(\tau)\rho_{\mathsf{A}}, \\
&A_0(\tau)\rho_{\mathsf{A}}\alpha_{\mathsf{A}}, \ldots, A_m(\tau)\rho_{\mathsf{A}}\alpha_{\mathsf{A}}, Z(\tau)\rho_{\mathsf{A}}\alpha_{\mathsf{A}}, \\
&B_0(\tau)\rho_{\mathsf{B}}, \ldots, B_m(\tau)\rho_{\mathsf{B}}, Z(\tau)\rho_{\mathsf{B}}, \\
&B_0(\tau)\rho_{\mathsf{B}}\alpha_{\mathsf{B}}, \ldots, B_m(\tau)\rho_{\mathsf{B}}\alpha_{\mathsf{B}}, Z(\tau)\rho_{\mathsf{B}}\alpha_{\mathsf{B}}, \\
&C_0(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}, \ldots, C_m(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}, Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}, \\
&C_0(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\alpha_{\mathsf{C}}, \ldots, C_m(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\alpha_{\mathsf{C}}, Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\alpha_{\mathsf{C}}, \\
&(A_0(\tau)\rho_{\mathsf{A}} + B_0(\tau)\rho_{\mathsf{B}} + C_0(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}})\beta, \ldots, \\
&\quad (A_m(\tau)\rho_{\mathsf{A}} + B_m(\tau)\rho_{\mathsf{B}} + C_m(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}})\beta, \\
&\quad (Z(\tau)\rho_{\mathsf{A}} + Z(\tau)\rho_{\mathsf{B}} + Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}})\beta, \\
&\alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \gamma, \gamma\beta, Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}, A_0(\tau)\rho_{\mathsf{A}}, \ldots, A_n(\tau)\rho_{\mathsf{A}}\Big) .
\end{aligned}
$$

2. Sample $\vec{\alpha}$ in $\mathbb{F}_r^8$ at random.
3. Compute $\mathsf{pp} := C(\vec{\alpha}) \cdot \mathcal{G}$.

4. Output pp.[7]

Fitting in the class $\mathbf{C}^{\mathsf{S}}$. We explain how to construct the above circuit C so to belong to the class $\mathbf{C}^{\mathsf{S}}$. We ensure the following: each gate of C is either a constant gate, or an addition gate that combines outputs of previous gates, or a multiplication gate whose two inputs are an output of a previous gate and a circuit input; these conditions satisfy the requirements of the class $\mathbf{C}^{\mathsf{S}}$ (see Section 3.3.4). Concretely, the outputs of C are computed as follows:

- The outputs $(1, \tau, \dots, \tau^d)$ are computed by a d-gate subcircuit that multiplies 1 by τ to obtain the output τ, then multiplies τ by τ to obtain the output τ^2, then multiples τ^2 by τ to obtain the output τ^3, and so on until the output τ^d.
- We invoke the results of Section 3.12.3 to obtain a subcircuit in $\mathbf{C}^{\mathsf{S}}$ that maps $1, \tau, \dots, \tau^d$ to $L_1(\tau), \dots, L_d(\tau)$, where $L_1, \dots, L_d$ are the Lagrange interpolation polynomials for certain suitably chosen evaluation points $\delta_1, \dots, \delta_d \in \mathbb{F}_r$.
- For $i = 0, \dots, m$, each of the values $A_i(\tau)$, $B_i(\tau)$ and $C_i(\tau)$ is a linear combination of $L_1(\tau), \dots, L_d(\tau)$ and can be computed by an addition gate. Note that these values are not part of the circuit output, however producing them does *not* produce non-public information. Indeed, each each $A_i(\tau)$, $B_i(\tau)$, and $C_i(\tau)$ is also a linear combination of $1, \tau, \dots, \tau^d$ so can be computed publicly from the circuit output. The only reason for invoking the results of Section 3.12.3 is efficiency: the terms $A_i(\tau)$, $B_i(\tau)$, and $C_i(\tau)$ are not sparse in the monomial basis $(1, \tau, \dots, \tau^d)$, but they are sparse in the Lagrange basis $(L_1(\tau), \dots, L_d(\tau))$.
- For $i = 0, \dots, m$, the output $A_i(\tau)\rho_{\mathsf{A}}$ is computed by a multiplication gate whose inputs are the previously computed value $A_i(\tau)$ and the input ρ_{A}. The same applies for the outputs $B_i(\tau)\rho_{\mathsf{B}}$. Similarly, the output $A_i(\tau)\rho_{\mathsf{A}}\alpha_{\mathsf{A}}$ is computed by a multiplication gate whose inputs are the previously computed value $A_i(\tau)\rho_{\mathsf{A}}$ and the input α_{A}. The same applies for the outputs $B_i(\tau)\rho_{\mathsf{B}}\alpha_{\mathsf{B}}$.
- The output $Z(\tau)$ is the degree d polynomial that vanishes on the set $\{\delta_1, \dots, \delta_d\}$; it is sparse in $(1, \tau, \dots, \tau^d)$, and can thus be computed by an addition gate that takes a suitable linear combination of these. As before, producing this auxiliary value does not produce any non-public information. The terms $Z(\tau)\rho_{\mathsf{A}}$ and $Z(\tau)\rho_{\mathsf{B}}$ are computed by multiplication gates whose inputs are the output $Z(\tau)$ and the input ρ_{A}, respectively, ρ_{B}. Similarly, the terms $Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$, $Z(\tau)\rho_{\mathsf{A}}\alpha_{\mathsf{A}}$, $Z(\tau)\rho_{\mathsf{B}}\alpha_{\mathsf{B}}$ are computed by multiplication

[7] The first $d + 7m + 15$ elements in pp form the *proving key* pk, while the remaining $n + 7$ form the *verification key* vk.

gates whose inputs are the outputs $Z(\tau)\rho_{\mathsf{A}}$ or, respectively, $Z(\tau)\rho_{\mathsf{B}}$ and the corresponding inputs ρ_{B}, α_{A} and α_{B}. Finally, $Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\alpha_{\mathsf{C}}$ is computed by a multiplication gate whose inputs are the output $Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$ and the input α_{C}.

- For $i = 0, \dots, m$, the output $C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\alpha_{\mathsf{C}}$ is computed by a multiplication gate whose inputs are the output $C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$ and the input α_{C}.
- For $i = 0, \dots, m$, the auxiliary value $A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$ is computed as the sum of the three outputs $A_i(\tau)\rho_{\mathsf{A}}$, $B_i(\tau)\rho_{\mathsf{B}}$, and $C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$, therefore this auxiliary value can be computed publicly. For $i = 0, \dots, m$, the output $(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}})\beta$ is computed by a multiplication gate whose inputs are the public value $A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$ and the input β. We take the same approach to compute $(Z(\tau)\rho_{\mathsf{A}} + Z(\tau)\rho_{\mathsf{B}} + Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}})\beta$.
- Each of the outputs α_{A},α_{B},α_{C}, and γ are computed by a multiplication gate whose inputs are the constant 1 and the respective input. Finally, the term $\gamma\beta$ is computed by a multiplication gate whose input is the previously computed output γ and the input β.

Our implementation realizes the above approach (and, in particular, we have ensured that the overall circuit lies in $\mathbf{C}^{\mathsf{S}}$).

As an important technical challenge, the outputs $C_0(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}, \dots, C_m(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}$ are not part of the CRS for the [PGHR13] proof system, and thus the original security proof does not apply. However, one can still prove the extended proof system secure, and we do so in Appendix A.

3.12.2 Example for a SSP-based zk-SNARK

We explain how the generator of [DFGK14]'s zk-SNARK can be cast as computing the encoding of a random evaluation of a certain circuit C that lies in $\mathbf{C}^{\mathsf{S}}$.

Supported NP relations. This zk-SNARK supports boolean circuit satisfiability (see Footnote 5), i.e., relations $\mathcal{R}_D = \{(\vec{x}, \vec{w}) \in \{0,1\}^n \times \{0,1\}^h : D(\vec{x}, \vec{w}) = 0^\ell\}$ where $D \colon \{0,1\}^n \times \{0,1\}^h \to \{0,1\}^\ell$ is a boolean circuit.

SSPs. The construction is based on *square span programs* (SSP) [DFGK14]: a SSP of size m and degree d over $\mathbb{F}$ is a tuple $(\vec{A}, Z)$, where $\vec{A}$ is a vector of $m+1$ polynomials in $\mathbb{F}^{\leq d-1}[z]$ and $Z \in \mathbb{F}[z]$ has degree exactly d. As shown in [DFGK14], each relation $\mathcal{R}_D$ can be reduced to a certain relation $\mathcal{R}_{(\vec{A},Z)}$, which captures "SSP satisfiability", by computing $(\vec{A}, Z) := \mathsf{GetSSP}(D)$ for a suitable function GetSSP; if D has N_{w} wires and N_{g} gates, then the resulting SSP has size $m = N_{\mathsf{w}}$ and degree $d \approx N_{\mathsf{w}} + N_{\mathsf{g}}$.

The parameter generator. On input a boolean circuit $D\colon \{0,1\}^n \times \{0,1\}^h \to \{0,1\}^\ell$, the generator does the following.

1. Compute $(\vec{A}, \vec{B}, \vec{C}, Z) := \mathsf{GetSSP}(D)$, and denote by m and d the SSP's size and degree; then construct an $\mathbb{F}_r$-arithmetic circuit $C\colon \mathbb{F}_r^3 \to \mathbb{F}_r^{d+2m+n+9}$ such that $C(\tau, \beta, \gamma)$ computes the following outputs:

$$\begin{aligned}\Big(&1, \tau, \dots, \tau^d,\\ &A_0(\tau), \dots, A_m(\tau), Z(\tau),\\ &A_0(\tau)\beta, \dots, A_m(\tau)\beta, Z(\tau)\beta,\\ &\gamma, \gamma\beta, Z(\tau), A_0(\tau), \dots, A_n(\tau)\Big) \ .\end{aligned}$$

2. Sample $\vec{\alpha}$ in $\mathbb{F}_r^3$ at random.
3. Compute $\mathsf{pp} := C(\vec{\alpha}) \cdot \mathcal{G}$.
4. Output pp.[8]

Fitting in the class $\mathbf{C}^{\mathsf{S}}$. We explain how to construct the above circuit C so to belong to the class $\mathbf{C}^{\mathsf{S}}$. We ensure the following: each gate of C is either a constant gate, or an addition gate that combines outputs of previous gates, or a multiplication gate whose two inputs are an output of a previous gate and a circuit input; these conditions satisfy the requirements of the class $\mathbf{C}^{\mathsf{S}}$ (see Section 3.3.4). Concretely, the outputs of C are computed as follows:

- The outputs $(1, \tau, \dots, \tau^d)$ are computed by a d-gate subcircuit that multiplies 1 by τ to obtain the output τ, then multiplies τ by τ to obtain the output τ^2, then multiples τ^2 by τ to obtain the output τ^3, and so on until the output τ^d.
- We invoke the results of Section 3.12.3 to obtain a subcircuit in $\mathbf{C}^{\mathsf{S}}$ that maps $1, \tau, \dots, \tau^d$ to $L_1(\tau), \dots, L_d(\tau)$, where $L_1, \dots, L_d$ are the Lagrange interpolation polynomials for certain suitably chosen evaluation points $\delta_1, \dots, \delta_d \in \mathbb{F}_r$.
- For $i = 0, \dots, m$, the output $A_i(\tau)$ is a linear combination of $L_1(\tau), \dots, L_d(\tau)$ and can be computed by an addition gate. (We recall that each $A_i(\tau)$ is also a linear combination of $1, \tau, \dots, \tau^d$, but it is not "sparse" in this basis.)
- The output $Z(\tau)$ is the degree d polynomial that vanishes on the set $\{\delta_1, \dots, \delta_d\}$; it is sparse in $(1, \tau, \dots, \tau^d)$, and can thus be computed by an addition gate that takes a suitable linear combination of these.
- For $i = 0, \dots, m$, the output $A_i(\tau)\beta$ is computed by a multiplication gate whose inputs

[8] The first $d + 2m + 5$ elements in pp form the *proving key* pk, while the remaining $n + 4$ form the *verification key* vk.

are the previously computed output $A_i(\tau)$ and the input β. The same applies for obtaining $Z(\tau)\beta$ from the output $Z(\tau)$ and the input β.

- The outputs γ and $\gamma\beta$ are computed via two multiplication gates: the first multiplies the constant 1 by the input γ, the second multiplies γ (the output of the previous gate) by the input β.

Our implementation realizes the above approach (and, in particular, we have verified that the overall circuit lies in $\mathbf{C}^{\mathsf{S}}$).

3.12.3 Circuits for polynomial interpolation

The two generators of two zk-SNARK constructions from Section 3.12.1 and Section 3.12.2 require computing a solution to a certain *polynomial interpolation problem*. The subcircuit computing performing polynomial interpolation is not straight-forward and the concrete choice of the subcircuit greatly affects the S-depth and size of the generator circuit.

This subsection is organized as follows. First, we describe the concrete interpolation problem, required for the both circuit generators mentioned above. Next, we argue that the core of the problem is computing a certain vector of Lagrange coefficients and provide two solutions for computing them. Finally, we summarize the cost characteristics of the two solutions in Figure 3.7.

The polynomial interpolation problem

The zk-SNARK generators in Section 3.12.1 and Section 3.12.2 require solving the following polynomial interpolation problem. The evaluation problem fixes a set W of evaluation points $\delta_1, \ldots, \delta_d \in \mathbb{F}_r$ and $m \cdot d$ values $y_{i,j} \in \mathbb{F}_r$ of m target polynomials $P_i(z)$. Each polynomial $P_i(z)$ has degree $d-1$ and satisfies $P_i(\delta_j) = y_{i,j}$ for $j = 1, \ldots, d$. By the Lagrange interpolation theorem each $P_i(z)$ is uniquely determined. The input to the problem is an interpolation point $\tau \in \mathbb{F}_r$ and the output is m interpolated values $(P_1(\tau), \ldots, P_m(\tau))$.

To summarize, we are given m polynomials $P_1, \ldots, P_m$, specified by their values at d evaluation points common to all polynomials, and seek to obtain a circuit in $\mathbf{C}^{\mathsf{S}}$ that, on input τ, outputs $P_i(\tau)$ $(1 \leq i \leq m)$. This problem is solved by using Lagrange interpolation. Let $L_j(z)$ be the j-th Lagrange interpolation polynomial, defined as the unique polynomial of degree at most $d-1$ satisfying $L_j(\delta_j) = 1$ and $L_j(\delta_i) = 0$ for all $i \neq j$. Then each polynomial P_i can be expressed as a weighted sum of Lagrange polynomials L_j as follows: $P_i(z) = \sum_{j=1}^{d} y_{i,j} \cdot L_j(z)$.

Thus, to evaluate all $P_i(\tau)$ it suffices to compute all Lagrange interpolants $L_j(\tau)$. Each $L_j(z)$ has an explicit form: $L_j(z) := \prod_{\substack{i=1\\ i\neq j}}^{d}(z-\delta_i)/(\delta_j-\delta_i)$. However, a straightforward evaluation of a single interpolant $L_j(\tau)$ requires a size $O(d)$ circuit and obtaining all d interpolants requires size $O(d^2)$ circuit. Alternatively, one could compute $L_1(\tau)$ and calculate the remaining interpolants as follows: $L_{j+1}(z) = L_j(z)\cdot(z-\delta_j)/(z-\delta_{j+1})\cdot\beta_j$, where $\beta_j \in \mathbb{F}_r$ is a constant.[9] However, this approach cannot be realized in $\mathbf{C}^{\mathsf{S}}$: the class of supported circuits can't compute multiplicative inverses.

Solving polynomial interpolation problem using FFTs

We use the fast Fourier transform (FFT) to efficiently compute the Lagrange interpolation polynomials $\{L_j(\tau)\}_{j=1}^{d}$ via a circuit in $\mathbf{C}^{\mathsf{S}}$. Let $P(z) := \sum_{i=0}^{d-1} a_i \cdot z^i$ be a degree $d-1$ polynomial with coefficients in $\mathbb{F}_r$, and let $W = \{\delta_1,\ldots,\delta_d\}$ be a subset of $\mathbb{F}_r$. The (fast) Fourier transform computes evaluations of P on entire set W; we denote the vector of such evaluations as $\mathsf{FFT}_W(a_0,\ldots,a_{d-1})$: $\mathsf{FFT}_W(a_0,\ldots,a_{d-1}) := (P(\delta_1),\ldots,P(\delta_{d-1}))$.

The fast Fourier transform is particularly efficient when the set W has multiplicative structure. In the zk-SNARK applications the set W is chosen to be the set of d-th roots of unity where d is a power of 2. More precisely, let ω be the d-th root of unity with $\omega^d = 1$ and $\omega^i \neq 1$ for $1 \leq i < d$. We set $\delta_i = \omega^i$ and then $W = \{1,\omega,\omega^2,\ldots,\omega^{d-1}\}$. For convenience we will extend the notation of $\mathsf{FFT}_\omega(a_0,\ldots,a_{d-1})$ to mean $\mathsf{FFT}_W(a_0,\ldots,a_{d-1})$ with W defined as $W := \{1,\omega,\omega^2,\ldots,\omega^{d-1}\}$.

We note that for the particular choice of W, the fast Fourier transform and the interpolation problem described above are related as follows:

$$\mathsf{FFT}_W(L_1(\tau),\ldots,L_d(\tau)) = (1,\tau,\tau^2,\ldots,\tau^{d-1}) .$$

Indeed, the j-th element of $\mathsf{FFT}_W(L_1(\tau),\ldots,L_d(\tau))$ equals $\sum_{i=0}^{d-1} L_{i+1}(\tau)\omega^{ij}$. Let $Q(z) := z^j$; the values of Q, when evaluated on W are $Q(\omega^i) = \omega^{ij}$ $(1 \leq i \leq d)$. Therefore, $Q(\tau)$ can be interpolated using Lagrange polynomials as follows: $\tau^j = Q(\tau) = \sum_{i=0}^{d-1} L_{i+1}(\tau)\omega^{ij}$. Note that this exactly matches the expression before.

We conclude that the Lagrange interpolants $L_j(\tau)$ can be computed by taking the *inverse* Fourier transform of the vector $(1,\tau,\tau^2,\ldots,\tau^{d-1})$. It is well known that the inverse of $\mathsf{FFT}_\omega(\cdot)$ is $\frac{1}{d}\mathsf{FFT}_{\omega^{-1}}(\cdot)$, therefore in the following discussion we will focus on computing

[9] More precisely, we have $\beta_j = \prod_{\substack{i=1\\ i\neq j}}^{d}(\delta_j-\delta_i)/\prod_{\substack{i=1\\ i\neq j+1}}^{d}(\delta_{j+1}-\delta_i)$.

the (forward) FFT: the circuits for computing the inverse FFT are just slight variations of their forward versions.

A constant depth FFT circuit

The first FFT circuit we describe is the standard "butterfly" circuit, as covered in standard algorithms textbooks [CLRS01]. Recall that FFT evaluates a polynomial $P(z) := \sum_{i=0}^{d-1} a_i \cdot z^i$ on a set of points W. We express P as the sum of its even and odd components: $P(z) := P_{\mathsf{even}}(z^2) + z \cdot P_{\mathsf{odd}}(z^2)$, where $P_{\mathsf{even}}(z) := \sum_{i=0}^{d/2-1} a_{2i} \cdot z^i$ and $P_{\mathsf{odd}}(z) := \sum_{i=0}^{d/2-1} a_{2i+1} \cdot z^i$. Note that $\omega^{2i} = \omega^{2(i+d/2)}$ and $\omega^{d/2} = -1$, therefore the even-odd decomposition yields the following evaluation strategy:

$$\begin{aligned}\mathsf{FFT}_\omega(a_0, a_1, \dots, a_{d-1}) &= \left(P(\omega^i)\right)_{i=0}^{d-1} = \left(P_{\mathsf{even}}(\omega^{2i}) + \omega^i P_{\mathsf{odd}}(\omega^{2i})\right)_{i=0}^{d-1} \\ &= (\alpha_0 + \omega^0 \beta_0, \dots, \alpha_{d-1} + \omega^{d-1}\beta_{d-1}, \\ &\qquad \alpha_0 - \omega^0 \beta_0, \dots, \alpha_{d-1} - \omega^{d-1}\beta_{d-1}),\end{aligned}$$

where $\vec{\alpha} := \mathsf{FFT}_{\omega^2}(a_0, a_2, \dots, a_{d-2})$ and $\vec{\beta} := \mathsf{FFT}_{\omega^2}(a_1, a_3, \dots, a_{d-1})$.

In more detail, we compute the FFT by a circuit of $\log d$ layers. The last layer computes $\mathsf{FFT}_\omega(\vec{a})$ by having d addition gates of the form form $\alpha_i \pm \omega^i \beta_i$. Each preceding layer implements the corresponding recursive evaluation of half-size FFTs, whereas the first layer refers to the input vector. Overall the circuit requires $d \cdot \log d$ addition gates; as the circuit does not contain any multiplication gates its S-depth is just 1. We remark that each intermediate value computed by this circuit is a linear combination of the terms $a_0, \dots, a_{d-1}$, therefore the use of this circuit does not produce any auxiliary outputs that can not be publicly computed from its inputs.

A linear size FFT circuit

The size of the "butterfly" FFT circuit is quasi-linear. For certain applications the $\log d$ factor is significant, so we describe an alternative FFT circuit of linear size. This reduction in size comes as a trade-off: S-depth of our circuit is logarithmic.

The "butterfly" FFT circuit works for any input $\vec{a}$; the circuit we now describe uses, in an essential way, the fact that Lagrange coefficients are computed as an inverse FFT of the geometric progression $(1, \tau, \tau^2, \dots, \tau^{d-1})$. That is, in Lagrange setting $a_i = \tau^i$, so $P(z) := \sum_{i=0}^{d} (\tau z)^i$. Moreover, $P_{\mathsf{odd}}(z) = \sum_{i=0}^{d} \tau^{2i+1} \cdot z^i = \tau \cdot \sum_{i=0}^{d} \tau^{2i} \cdot z^i = \tau \cdot P_{\mathsf{even}}(z)$.

	Constant depth circuit of Section 3.12.3	Linear size circuit of Section 3.12.3
$\mathsf{depth_S}(C)$	1	$\log d$
#gates(C)	$d \cdot (\log d + 1)$	$5d - 4$
#const-gates(C)	1	1
#add-gates(C)	$d \cdot \log d$	$2d - 2$
#mul-gates(C)	$d - 1$	$3d - 3$
#wires(C)	$3(\log d + 1)d - 1$	$3(\log d + 1)d - 1$

Figure 3.7: S-depth, gate and wire counts of the subcircuit C in $\mathbf{C}^{\mathsf{S}}$ computing Lagrange coefficients for various choices of C. All measurements expressed as a function of d, the number of Lagrange evaluation points.

Therefore, we proceed as in "butterfly" construction described above, but instead of computing $\vec{\beta}$ recursively, we eliminate the recursive call and compute as $\vec{\beta} = \tau \cdot \vec{\alpha}$. More precisely, we let $\vec{\alpha} := \mathsf{FFT}_{\omega^2}\left(1, \tau^2, \ldots, \tau^{d-2}\right)$ be the "half-size" FFT on the even coefficients and compute the "full-size" FFT as follows:

$$\mathsf{FFT}_{\omega}\left(1, \tau, \tau^2, \ldots, \tau^{d-1}\right) = (\alpha_0 + \omega^0 \tau \alpha_0,\ \ldots,\ \alpha_{d/2-1} + \omega^{d/2-1} \tau \alpha_{d-1},$$
$$\alpha_0 - \omega^0 \tau \alpha_0,\ \ldots,\ \alpha_{d/2-1} - \omega^{d/2-1} \tau \alpha_{d/2-1})\ .$$

Note that each recursive call, the vector $(1, \tau^2, \ldots, \tau^{d-2})$ is a geometric progression, so by induction in each recursive call P_{odd} is a scaled copy of P_{even}.

Figure shows how this approach is realized as a circuit in $\mathbf{C}^{\mathsf{S}}$. As before, we compute the FFT by a $\log d$ layer circuit. The last layer refers to the preceding layer for the evaluations α_i of the "half-size" FFT computation. A sequence of $d/2$ multiplication gates perform the scaling by τ and d addition gates compute the "full-size" FFT output. Similarly, the next-to-last layer contains $d/4$ multiplication gates that perform scaling by τ^2 and $d/2$ addition gates for computing that layer's output, and so on. The first layer refers to constant 1 as the base case output and its multiplication gate performs scaling by $\tau^{d/2}$. Overall, the circuit contains $2d - 2$ addition gates and $d - 1$ multiplication gates. Finally, we remark that similar to the butterfly circuit above, all intermediate values of this FFT circuit are just linear combinations of $1, \tau, \ldots, \tau^{d-1}$ and therefore its use does not produce any auxiliary outputs that can not be publicly computed from its inputs.

Appendix A

Proofs of security

Chapter 3 describes a multiparty computation protocol for sampling common reference string of many zero-knowledge SNARK protocols. For technical reasons this protocol assumes that the zk-SNARK constructions remain secure even when the CRS sampling procedure outputs extra elements. More precisely, many zk-SNARK constructions from the literature obtain their parameters by sampling from the distribution $\{C(\tau) \cdot \mathcal{P} \,:\, \tau \leftarrow \mathbb{F}^n\}$, where the circuit C does not belong to the sampling circuit class $\mathcal{C}^{\mathsf{S}}$. The protocol described in Chapter 3 thus considers augmented circuits C' which, compared to original sampling circuits C, produce additional outputs, but fits in the class $\mathcal{C}^{\mathsf{S}}$. In this appendix we prove that, for proof systems of Danezis et al. [DFGK14] and Parno et al. [PGHR13], sampling from such expanded distribution $\{C'(\tau) \cdot \mathcal{P} \,:\, \tau \leftarrow \mathbb{F}^n\}$ does not break security of the underlying zk-SNARK proof systems.

A.1 Preliminaries

Fields and polynomials. We denote by $\mathbb{F}$ a finite field and by $\mathbb{F}_r$ the field of size r. We denote by $\mathbb{F}[z]$ the ring of univariate polynomials over $\mathbb{F}$, and by $\mathbb{F}^{\leq d}[z]$ the subring of univariate polynomials of degree at most d. For a polynomial $P \in \mathbb{F}[z]$ we write $(P)_i$ to denote the coefficient of z^i in P, so $P(z)$ can be expanded as $\sum_{i=0}^{\deg P} (P)_i z^i$. When S is a set of polynomials over $\mathbb{F}$, we denote by $\mathrm{span}(S)$ the set of all $\mathbb{F}$-linear combinations of polynomials in S.

Notation for extractors. We write $(y; x) \leftarrow (\mathcal{A} \| \mathcal{E}_{\mathcal{A}})(z)$ when the algorithm $\mathcal{A}$, given input z, outputs y, and the algorithm $\mathcal{E}_{\mathcal{A}}$, given the same input (including the same random tape as $\mathcal{A}$), outputs x.

A useful lemma on spans of polynomials. For ensuring that a polynomial is in a certain span, we use [GGPR13, Lemma 10], which we restate and for which we provide an explicit proof.

Lemma A.1.1. *Let $d \in \mathbb{N}$ and let $S = \{p_1(z), \ldots, p_n(z)\} \subseteq \mathbb{F}^{\leq d}[z]$ be a set of polynomials. Let $a(z) \in \mathbb{F}^{\leq d+1}[z]$ be chosen uniformly at random, subject to $(a(z) \cdot p_k(z))_{d+1} = 0$ for $k = 1, \ldots, n$. Then for all algorithms $\mathcal{A}$:*

$$\Pr\left[\begin{array}{c} u(z) \in \mathbb{F}^{\leq d}[z] \\ u(z) \notin \mathrm{span}(S) \\ (a(z) \cdot u(z))_{d+1} \neq 0 \end{array} \middle| \begin{array}{c} \tau \leftarrow \mathbb{F} \\ u(z) \leftarrow \mathcal{A}(S, \tau, a(\tau)) \end{array}\right] \leq 1/|\mathbb{F}| \ .$$

Proof. Let $a(z) = \sum_{i=0}^{d+1} a_i z^i$ and express the polynomials $p_k(z)$ in similar fashion: $S = \{\sum_{i=0}^{d} p_{k,i} z^i\}_k$. To the adversary, $a(z)$ is a uniformly random polynomial subject to $M \cdot \vec{a} = \vec{b}$, where

$$M := \begin{pmatrix} p_{1,0} & \cdots & p_{1,d} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ p_{n,0} & \cdots & p_{n,d} & 0 \\ \tau^{d+1} & \ldots & \tau & 1 \end{pmatrix}, \ \vec{a} := \begin{pmatrix} a_{d+1} \\ \vdots \\ a_0 \end{pmatrix}, \text{ and } \vec{b} := \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a(\tau) \end{pmatrix} .$$

The rows of M encode the constraints $(a(z) \cdot p_k(z))_{d+1} = 0$, and the knowledge of $a(\tau)$, respectively. We may assume, without loss of generality, that $\dim(\mathrm{span}(S)) < d + 1$, because if the polynomials in S spanned a $(d+1)$-dimensional space then any polynomial $u(z)$ output by $\mathcal{A}$ would be in $\mathrm{span}(S)$ and the lemma would follow. Therefore it suffices to consider the case that M is not full rank, and there exist $l = (d+2) - \mathrm{rank}(M) > 0$ row vectors $v_1, \ldots, v_l$ that augment M to a full rank matrix M'.

Consider the following system of equations:

$$M' \cdot \vec{a} = \begin{bmatrix} M \\ \hline v_1 \\ \vdots \\ v_l \end{bmatrix} \cdot \vec{a} = \begin{pmatrix} b \\ \hline c_1 \\ \vdots \\ c_l \end{pmatrix} .$$

The matrix M' is full rank so each choice of $(c_1, \ldots, c_l) \in \mathbb{F}^l$ corresponds to a unique $\vec{a}$ satisfying the system above. Thus picking a random $a(z)$ subject to $(a(z) \cdot p_k(z))_{d+1} = 0$ and a fixed value of $a(\tau)$ is the same as picking random $c_1, \ldots, c_l$, and solving for the coefficients a_i of $a(z)$.

Picking $c_1, \dots, c_l$ can be done after the adversary returns the polynomial $u(z) = \sum_{i=0}^{d} u_i z^i$ as this does not change the view of the adversary. Express $(u_0, \dots, u_d, 0)$ as a linear combination of the rows of M'. The polynomial $u(x)$ is not in $\mathrm{span}(S)$, therefore this linear combination has non-zero coefficient for at least one of $v_1, \dots, v_l$. We conclude that the value of $(a(z) \cdot u(z))_{d+1} = (u_0, \dots, u_d, 0) \cdot \vec{a}$ has a contribution from random element c_i for some i, and is thus uniformly random. □

Remark A.1.2. It would be useful to prove a stronger version of the theorem, letting $a(z)$ be of as low degree as possible, and still having $(a(z) \cdot p_k(z))_{d+1} = 0$ imply that $(a(z) \cdot u(z))_{d+1} = 0$. Unfortunately, even if the adversary does not get to see $a(\tau)$, degree $d+1$ is necessary unless more is known about the set S. For example, if we knew that $\dim(\mathrm{span}(S)) < d-1$, which is true when $|S| < d-1$, we could prove the theorem for $a(z) \in \mathbb{F}^{\leq d}[z]$.

A.1.1 Security assumptions

We introduce several security assumptions:

Assumption A.1.3 (q-PKE). *The $q(\lambda)$-power knowledge of exponent assumption holds relative to a bilinear group generator $\mathcal{G}$ for the class $\mathcal{Z}$ if, for every polynomial-size auxiliary input generator $Z \in \mathcal{Z}$, and every polynomial-size adversary $\mathcal{A}$, there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$ such that:*

$$\Pr\left[\begin{array}{c|c} \begin{array}{c} e(\mathcal{Q}_1, \alpha \cdot \mathcal{P}_2) = e(\mathcal{P}_1, \mathcal{Q}_2) \\ \textit{and} \\ \mathcal{Q}_1 \neq \sum_{i=0}^{q} a_i \tau^i \cdot \mathcal{P}_1 \end{array} & \begin{array}{c} \mathsf{gk} := (r, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathcal{G}(1^\lambda) \\ \mathcal{P}_1 \leftarrow \mathbb{G}_1, \mathcal{P}_2 \leftarrow \mathbb{G}_2, \tau \leftarrow \mathbb{F}_r, \alpha \leftarrow \mathbb{F}_r \\ z \leftarrow Z(\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \tau) \\ \sigma := \begin{pmatrix} \mathsf{gk} & \mathcal{P}_1 & \tau \cdot \mathcal{P}_1 & \dots & \tau^q \cdot \mathcal{P}_1 \\ & \mathcal{P}_2 & \tau \cdot \mathcal{P}_2 & \dots & \tau^q \cdot \mathcal{P}_2 \\ & \alpha \cdot \mathcal{P}_1 & \alpha\tau \cdot \mathcal{P}_1 & \dots & \alpha\tau^q \cdot \mathcal{P}_1 \\ & \alpha \cdot \mathcal{P}_2 & \alpha\tau \cdot \mathcal{P}_2 & \dots & \alpha\tau^q \cdot \mathcal{P}_2 \end{pmatrix} \\ (\mathcal{Q}_1, \mathcal{Q}_2; a_0, \dots, a_q) \leftarrow (\mathcal{A} \| \mathcal{E}_{\mathcal{A}})(\sigma, z) \end{array} \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

Assumption A.1.4 (q-PDH). *The $q(\lambda)$-power Diffie-Hellman assumption holds relative to a* *bilinear group generator $\mathcal{G}$ if, for every polynomial-size adversary $\mathcal{A}$:*

$$\Pr\left[\ \mathcal{Q} = \tau^{q+1}\cdot\mathcal{P}_1 \ \middle|\ \begin{array}{c} \mathsf{gk} := (r,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e) \leftarrow \mathcal{G}(1^\lambda) \\ \mathcal{P}_1 \leftarrow \mathbb{G}_1, \mathcal{P}_2 \leftarrow \mathbb{G}_2, \tau \leftarrow \mathbb{F}_r \\ \sigma := \begin{pmatrix} \mathsf{gk} & \mathcal{P}_1 & \tau\cdot\mathcal{P}_1 & \ldots & \tau^q\cdot\mathcal{P}_1 & \tau^{q+2}\cdot\mathcal{P}_1 & \ldots & \tau^{2q}\cdot\mathcal{P}_1 \\ & \mathcal{P}_2 & \tau\cdot\mathcal{P}_2 & \ldots & \tau^q\cdot\mathcal{P}_2 & \tau^{q+2}\cdot\mathcal{P}_2 & \ldots & \tau^{2q}\cdot\mathcal{P}_2 \end{pmatrix} \\ \mathcal{Q} \leftarrow \mathcal{A}(\sigma) \end{array} \right] \leq \mathsf{negl}(\lambda) \ .$$

Assumption A.1.5 (q-TSDH). *The $q(\lambda)$-power target group strong Diffie-Hellman assumption holds relative to a bilinear group generator $\mathcal{G}$ if, for every polynomial-size adversary $\mathcal{A}$:*

$$\Pr\left[\ \begin{array}{c} c \in \mathbb{Z}_r \setminus \{\tau\} \\ \textit{and} \\ Y = e(\mathcal{P}_1,\mathcal{P}_2)^{\frac{1}{\tau-c}} \end{array} \ \middle|\ \begin{array}{c} \mathsf{gk} := (r,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e) \leftarrow \mathcal{G}(1^\lambda) \\ \mathcal{P}_1 \leftarrow \mathbb{G}_1, \mathcal{P}_2 \leftarrow \mathbb{G}_2, \tau \leftarrow \mathbb{F}_r \\ \sigma := \begin{pmatrix} \mathsf{gk} & \mathcal{P}_1 & \tau\cdot\mathcal{P}_1 & \ldots & \tau^q\cdot\mathcal{P}_1 \\ & \mathcal{P}_2 & \tau\cdot\mathcal{P}_2 & \ldots & \tau^q\cdot\mathcal{P}_2 \end{pmatrix} \\ (c,Y) \leftarrow \mathcal{A}(\mathsf{gk},\sigma) \end{array} \right] \leq \mathsf{negl}(\lambda) \ .$$

The q-PKE assumption was originally introduced by Groth [Gro10] in the symmetric setting. Variants of the q-PKE assumption are used in various papers: [PGHR13] uses the symmetric variant from [Gro10]. [DFGK14] and [CFH+15] both introduce and use asymmetric variants, slightly different from one we use. When $\mathbb{G}_1 = \mathbb{G}_2$, the variant presented here is equivalent to the symmetric assumption of [Gro10].

The q-PDH assumption was originally introduced by Groth [Gro10] in the symmetric setting. Variants of the q-PDH assumption are used in various papers: [PGHR13] uses the symmetric variant from [Gro10], [DFGK14] introduces and uses the asymmetric variant described above, [CFH+15] uses the same asymmetric variant.

The q-SDH assumption was originally introduced by Boneh and Boyen [BB04] where the adversary is required to output a pair $(c, \frac{1}{\tau-c}\cdot\mathcal{P}_1)$. The weaker q-TSDH assumption in the *target group* was introduced and used by [PGHR13]. Variants of the q-TSDH assumption are used in various papers: [DFGK14] introduces and uses the asymmetric variant described above, [CFH+15] uses the same asymmetric variant.

A.1.2 Duplex pairing groups

A group $\mathbb{G}$ of prime order r is *duplex pairing* if there are order-r groups $\mathbb{G}_1$ and $\mathbb{G}_2$ such that (i) there is a pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for some target group $\mathbb{G}_T$, and (ii) there are generators $\mathcal{P}_1$ of $\mathbb{G}_1$ and $\mathcal{P}_2$ of $\mathbb{G}_2$ such that $\mathbb{G}$ is isomorphic to $\{(\alpha\mathcal{P}_1, \alpha\mathcal{P}_2) \mid \alpha \in \mathbb{F}_r\} \subseteq \mathbb{G}_1 \times \mathbb{G}_2$. We can use e to endow $\mathbb{G}$ with a pairing $e_{\mathbb{G}} \colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ as follows. For any two elements $\mathcal{Q} = (\mathcal{Q}_1, \mathcal{Q}_2) \in \mathbb{G}_1 \times \mathbb{G}_2$ and $\mathcal{R} = (\mathcal{R}_1, \mathcal{R}_2) \in \mathbb{G}_1 \times \mathbb{G}_2$ of $\mathbb{G}$ we define $e_{\mathbb{G}}(\mathcal{Q}, \mathcal{R}) := e(\mathcal{Q}_1, \mathcal{R}_2) = e(\mathcal{R}_1, \mathcal{Q}_2)$, where the last equality holds as $\mathbb{G}$ is duplex-pairing.

A.2 The DFGK zk-SNARK protocol

Definition A.2.1. *A* ***square span program (SSP)*** *of size m and degree d over a field $\mathbb{F}$ is a pair $(\vec{v}, t)$, where $\vec{v}$ is a vector of $m+1$ polynomials $v_0(z), \ldots v_m(z) \in \mathbb{F}^{\leq d-1}[z]$ and $t(z) \in \mathbb{F}[z]$ is a monic polynomial of degree exactly d.*

Definition A.2.2. *The* ***satisfaction problem*** *of a size-m SSP $(\vec{v}, t)$ is the relation $\mathcal{R}_{(\vec{v},t)}$ of pairs $(\vec{x}, \vec{a}) \in \mathbb{F}^n \times \mathbb{F}^m$ satisfying the following conditions: (a) $n \leq m$ and $x_i = a_i$ for $1 \leq i \leq n$ (that is, $\vec{a}$ extends $\vec{x}$); and (b) $t(z)$ divides $(v_0(z) + \sum_{i=1}^{m} a_i v_i(z))^2 - 1$.*

Remark A.2.3. The protocol in Figure A.1 is [DFGK14]'s zk-SNARK, presented with minor modifications. First, instead of having the generator G output a single common reference string crs, we partition its components in the proving key pk and the verification key vk. Second, for efficiency reasons we augment vk with the elements $\mathsf{vk}_{\mathsf{IC},i} := v_i(\tau) \cdot \mathcal{P}_1$ and $\mathsf{vk}_{\alpha t} := \alpha t(\tau) \cdot \mathcal{P}_2$. Doing so does not compromise security as $\mathsf{vk}_{\mathsf{IC},i}$ and $\mathsf{vk}_{\alpha t}$ are known linear combinations of public elements $\mathsf{pk}_{\mathsf{H},i}^{(1)} := \tau^i \cdot \mathcal{P}_1$, respectively, $\mathsf{pk}_{\mathsf{H},i}^{(2)} := \alpha\tau^i \cdot \mathcal{P}_2$ included in the proving key.

Remark A.2.4. As compared to Figure A.1, the implementation in **libsnark** has matching vk. The pk of **libsnark** has the following differences: (a) it additionally includes the terms $t(\tau) \cdot \mathcal{P}_1$, $\{v_i(\tau) \cdot \mathcal{P}_1\}_{i=n+1}^{m}$ and $\{v_i(\tau) \cdot \mathcal{P}_2\}_{i=0}^{m}$; (b) it omits the terms $\mathsf{pk}_{\mathsf{H},i}^{(2)}$, as they are not needed by the prover.

Public parameters. The group key $\mathsf{gk} := (r, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathcal{G}(1^\lambda)$: a prime r, two cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order r, and a pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ (where $\mathbb{G}_T$ is also cyclic of order r).

(a) Key generator G

- INPUTS: square span program $(\vec{v}, t)$
- OUTPUTS: proving key pk, verification key vk, simulation trapdoor $\mathsf{trap}_\mathcal{S}$ and extraction trapdoor $\mathsf{trap}_\mathcal{E}$

1. Sample two generators $\mathcal{P}_1 \in \mathbb{G}_1$, $\mathcal{P}_2 \in \mathbb{G}_2$ and four field elements $\alpha, \beta, \tau, \rho \in \mathbb{F}_r$, all at random.
2. Set $\mathsf{pk} := (\mathsf{gk}, (\vec{v}, t), \mathsf{pk}_{\beta t}, \mathsf{pk}_\mathsf{H}^{(1)}, \mathsf{pk}_\mathsf{H}^{(2)}, \mathsf{pk}_\mathsf{K})$ where
$$\begin{aligned} \mathsf{pk}_{\beta t} &:= \beta t(\tau) \cdot \mathcal{P}_1 , \\ \mathsf{pk}_\mathsf{H}^{(1)} &:= (\mathcal{P}_1, \dots, \tau^d \cdot \mathcal{P}_1) , \\ \mathsf{pk}_\mathsf{H}^{(2)} &:= (\alpha \cdot \mathcal{P}_2, \dots, \alpha\tau^d \cdot \mathcal{P}_2) , \\ \mathsf{pk}_\mathsf{K} &:= (\beta v_{n+1}(\tau) \cdot \mathcal{P}_1, \dots, \beta v_m(\tau) \cdot \mathcal{P}_1) . \end{aligned}$$
3. Set $\mathsf{vk} := (\mathsf{gk}, \mathsf{vk}_1, \mathsf{vk}_\alpha, \mathsf{vk}_{\alpha t}, \mathsf{vk}_\rho, \mathsf{vk}_{\rho\beta}, \mathsf{vk}_\mathsf{IC})$ where
$$\begin{matrix} \mathsf{vk}_1 := \mathcal{P}_1 & \mathsf{vk}_\alpha := \alpha \cdot \mathcal{P}_2 & \mathsf{vk}_{\alpha t} := \alpha t(\tau) \cdot \mathcal{P}_2 \\ \mathsf{vk}_\rho := \rho \cdot \mathcal{P}_2 & \mathsf{vk}_{\rho\beta} := \rho\beta \cdot \mathcal{P}_2 & \end{matrix} ,$$
and
$$\mathsf{vk}_\mathsf{IC} := (v_0(\tau) \cdot \mathcal{P}_1, \dots, v_n(\tau) \cdot \mathcal{P}_1) .$$
4. Set $\mathsf{trap}_\mathcal{S} := (\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \alpha, \beta, \tau)$.
5. Set $\mathsf{trap}_\mathcal{E} := (\mathcal{P}_2, \dots, \tau^d \cdot \mathcal{P}_2, \alpha \cdot \mathcal{P}_1, \dots, \alpha\tau^d \cdot \mathcal{P}_1)$.
6. Output $(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_\mathcal{S}, \mathsf{trap}_\mathcal{E})$.

(b) Simulator S

- INPUTS: simulation trapdoor $\mathsf{trap}_\mathcal{S}$, input $\vec{x} \in \mathbb{F}_r^n$
- OUTPUTS: simulated proof π

1. Sample $\phi \in \mathbb{F}_r$ at random.
2. Compute $h \in \mathbb{F}_r$ as follows:
$$h := \frac{(v_0(\tau) + \sum_{i=1}^n x_i v_i(\tau) + \phi)^2 - 1}{t(\tau)} ,$$
and set:
$$\begin{aligned} &\pi_\mathsf{H} := h \cdot \mathcal{P}_1,\ \pi_w := \phi \cdot \mathcal{P}_1,\ \pi_{w\beta} := \beta\phi \cdot \mathcal{P}_1, \\ &\pi_V^{(2)} := \alpha \left(v_0(\tau) + \sum_{i=1}^n x_i v_i(\tau) + \phi \right) \cdot \mathcal{P}_2 . \end{aligned}$$
3. Output $\pi := (\pi_\mathsf{H}, \pi_w, \pi_{w\beta}, \pi_V^{(2)})$.

(c) Prover P

- INPUTS: proving key pk, input $\vec{x} \in \mathbb{F}_r^n$, and assignment $\vec{a} \in \mathbb{F}_r^m$, where $(\vec{x}, \vec{a}) \in \mathcal{R}_{(\vec{v},t)}$.
- OUTPUTS: proof π

1. Sample $\delta \in \mathbb{F}_r$ at random.
2. Compute $\vec{h} = (h_0, \dots, h_0) \in \mathbb{F}_r^{d+1}$, the coefficients of the polynomial
$$H(z) := \frac{(v_0(z) + \sum_{i=1}^m a_i v_i(z) + \delta t(z))^2 - 1}{t(z)} .$$
3. Use $\mathsf{pk}_\mathsf{H}^{(1)}$ and $\mathsf{pk}_\mathsf{H}^{(2)}$ to compute
$$\begin{aligned} \pi_w &:= \left(\sum_{i=n+1}^m a_i v_i(\tau) + \delta t(\tau) \right) \cdot \mathcal{P}_1 , \\ \pi_V^{(2)} &:= \alpha \left(v_0(\tau) + \sum_{i=1}^n a_i v_i(\tau) + \delta t(\tau) \right) \cdot \mathcal{P}_2 \end{aligned}$$
4. Compute
$$\begin{aligned} \pi_{w\beta} &:= \sum_{i=n+1}^m a_i \cdot \mathsf{pk}_{\mathsf{K},i-n} + \delta \cdot \mathsf{pk}_{\beta t} \\ &= \beta \left(\sum_{i=n+1}^m a_i v_i(\tau) + \delta t(\tau) \right) \cdot \mathcal{P}_1 \\ \pi_\mathsf{H} &:= \sum_{i=0}^d h_i \cdot \mathsf{pk}_{\mathsf{H},i}^{(1)} = H(\tau) \cdot \mathcal{P}_1 . \end{aligned}$$
5. Output $\pi := (\pi_\mathsf{H}, \pi_w, \pi_{w\beta}, \pi_V^{(2)})$.

(d) Verifier V

- INPUTS: verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π
- OUTPUTS: decision bit b

1. Compute $\pi_V^{(1)} := (\mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^n x_i \cdot \mathsf{vk}_{\mathsf{IC},i}) + \pi_w$.
2. Check that $(\pi_V^{(1)}, \pi_V^{(2)})$ is a knowledge commitment:
$$e(\pi_V^{(1)}, \mathsf{vk}_\alpha) = e(\mathsf{vk}_1, \pi_V^{(2)}) .$$
3. Check that the same coefficients were used:
$$e(\pi_w, \mathsf{vk}_{\rho\beta}) = e(\pi_{w\beta}, \mathsf{vk}_\rho) .$$
4. Check SSP divisibility:
$$e(\pi_V^{(1)}, \pi_V^{(2)}) = e(\pi_\mathsf{H}, \mathsf{vk}_{\alpha t}) \cdot e(\mathsf{vk}_1, \mathsf{vk}_\alpha) .$$
5. If all checks pass, output $b := 1$ (accept), otherwise output $b := 0$ (reject).

Figure A.1: The zk-SNARK protocol of Danezis et al. [DFGK14].

Definition A.2.5. *Define the class $\mathcal{Z}_{\mathsf{SSP}}$ of auxiliary input generators as follows. An auxiliary input generator $Z_{(\vec{v},t)} \in \mathcal{Z}_{\mathsf{SSP}}$ on input $(\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \tau)$ samples random $\rho, \beta \in \mathbb{F}_r$ and returns the quadruple $(\{\beta v_i(\tau) \cdot \mathcal{P}_1\}_{i=n+1}^{m}, \beta t(\tau) \cdot \mathcal{P}_1, \rho \cdot \mathcal{P}_2, \rho\beta \cdot \mathcal{P}_2)$.*

Theorem A.2.6. *Assume that, relative to bilinear group generator $\mathcal{G}$, the $d(\lambda)$-PDH and $d(\lambda)$-TSDH assumptions hold, and $d(\lambda)$-PKE assumption holds for the class $\mathcal{Z}_{\mathsf{SSP}}$. Then the proof system in Figure A.1 is proof-of-knowledge. That is, for every security function $\lambda \colon \mathbb{N} \to \mathbb{N}$, every square span program $(\vec{v}, t)$ of degree $d(\lambda)$, and every polynomial-size adversary $\mathcal{A}$, there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$ such that:*

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk}, \vec{x}, \pi) = 1 & \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda) \\ \textit{and} & (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{S}}, \mathsf{trap}_{\mathcal{E}}) \leftarrow G(\mathsf{gk}, (\vec{v}, t)) \\ (\vec{x}, \vec{a}) \notin \mathcal{R}_{(\vec{v},t)} & ((\vec{x}, \pi); \vec{a}) \leftarrow (\mathcal{A}(\mathsf{pk}, \mathsf{vk}) \| \mathcal{E}_{\mathcal{A}}(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{E}})) \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

We prove the theorem by introducing and proving the three following lemmas.

Lemma A.2.7. *Assume that, $d(\lambda)$-PKE assumption holds for the class $\mathcal{Z}_{\mathsf{SSP}}$ relative to a bilinear group generator $\mathcal{G}$. Then, for every security function $\lambda \colon \mathbb{N} \to \mathbb{N}$, every square span program ($\vec{v}$, t) of degree $d(\lambda)$, and every polynomial-size adversary $\mathcal{A}$, there exists a polynomial-size adversary $\mathcal{A}'$ such that:*

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk}, \vec{x}, \pi) = 0 & \\ \textit{or all of the following hold:} & \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda) \\ \vec{x}' = \vec{x} & (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{S}}, \mathsf{trap}_{\mathcal{E}}) \leftarrow G(\mathsf{gk}, (\vec{v}, t)) \\ \pi' = \pi = (\pi_{\mathsf{H}}, \pi_w, \pi_{w\beta}, \pi_V^{(2)}) & ((\vec{x}, \pi); (\vec{x}', \pi', \delta, v(z))) \leftarrow (\mathcal{A}(\mathsf{pk}, \mathsf{vk}) \| \mathcal{A}'(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{E}})) \\ \pi_V^{(2)} = \alpha(v(\tau) + \delta t(\tau)) \cdot \mathcal{P}_2 & \\ \deg v(z) \leq d - 1 & \end{array}\right] \geq 1 - \mathsf{negl}(\lambda) \ ,$$

where α, τ and $\mathcal{P}_2$ above refer to the elements sampled by G.

In other words, whenever $\mathcal{A}$ outputs an accepting instance-proof pair, the adversary $\mathcal{A}'$ also returns the same instance-proof pair together with an "explanation" of the term $\pi_V^{(2)}$. We use the term "augmented adversary" to denote adversaries $\mathcal{A}'$ which produce such augmented output.

Proof. We use the SSP adversary $\mathcal{A}$ to construct an adversary $\mathcal{B}_{\mathsf{PKE}}$ for the d-PKE assumption, and use the PKE extractor for $\mathcal{B}_{\mathsf{PKE}}$ to obtain δ and $v(z)$. Essentially, $\mathcal{B}_{\mathsf{PKE}}$ calls adversary $\mathcal{A}$, and when the adversary returns proof π, the adversary $\mathcal{B}_{\mathsf{PKE}}$ computes and outputs a knowledge commitment $(\pi_V^{(1)}, \pi_V^{(2)})$. The d-PKE assumption guarantees an algorithm $\mathcal{E}_{\mathcal{B}_{\mathsf{PKE}}}$, the PKE extractor, which given the same inputs as $\mathcal{B}_{\mathsf{PKE}}$, returns a linear combination "explaining" the knowledge commitment. Our augmented adversary $\mathcal{A}'$ will read off δ and $v(z)$ from $\mathcal{E}_{\mathcal{B}_{\mathsf{PKE}}}$'s output. Despite the simplicity of the high level idea, we have to be careful to ensure that the view of $\mathcal{B}_{\mathsf{PKE}}$ is constructible given just the d-PKE challenge and auxiliary input; we do so as explained next.

Assume that d-PKE assumption holds for the class $\mathcal{Z}_{\mathsf{SSP}}$ and consider an adversary $\mathcal{B}_{\mathsf{PKE}}$ that works as follows. On input $\sigma := (\mathsf{gk}, \mathcal{P}_1, \tau \cdot \mathcal{P}_1, \ldots, \tau^d \cdot \mathcal{P}_1, \mathcal{P}_2, \tau \cdot \mathcal{P}_2, \ldots, \tau^d \cdot \mathcal{P}_2,$ $\alpha \cdot \mathcal{P}_1, \alpha\tau \cdot \mathcal{P}_1, \ldots, \alpha\tau^d \cdot \mathcal{P}_1, \alpha \cdot \mathcal{P}_2, \alpha\tau \cdot \mathcal{P}_2, \ldots, \alpha\tau^d \cdot \mathcal{P}_2)$, and auxiliary input $z := Z(\mathsf{gk}, \mathcal{P}_1,$ $\mathcal{P}_2, \tau)$, the adversary $\mathcal{B}_{\mathsf{PKE}}$:

1. Uses σ and z to construct the SSP keypair $(\mathsf{pk}, \mathsf{vk})$ for the square span program $(\vec{v}, t)$. Except for $\mathsf{vk}_{\mathsf{IC}}$ and $\mathsf{vk}_{\alpha t}$, every term of pk and vk appears verbatim in σ or z. $\mathcal{B}_{\mathsf{PKE}}$ constructs the terms $\mathsf{vk}_{\mathsf{IC},i} := v_i(\tau) \cdot \mathcal{P}_1$ and $\mathsf{vk}_{\alpha t} := \alpha t(\tau) \cdot \mathcal{P}_2$ as a linear combination of elements in its challenge; $\mathcal{B}_{\mathsf{PKE}}$ can always do so as $\deg v_i(z), \deg t(z) \leq d$.
2. Runs the SSP adversary $\mathcal{A}$ on $(\mathsf{pk}, \mathsf{vk})$. By construction $(\mathsf{pk}, \mathsf{vk})$ has the same distribution as the keypair output by the SNARK generator $G(\mathsf{gk}, (\vec{v}, t))$.
3. When $\mathcal{A}$ returns the instance-proof pair $(\vec{x}, \pi)$, $\mathcal{B}_{\mathsf{PKE}}$ parses the proof as $\pi = (\pi_{\mathsf{H}}, \pi_w,$ $\pi_{w\beta}, \pi_V^{(2)})$, computes $\pi_V^{(1)} := (\mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^n x_i \cdot \mathsf{vk}_{\mathsf{IC},i}) + \pi_w$ and returns the pair $(\pi_V^{(1)},$ $\pi_V^{(2)})$.

Whenever $\mathcal{A}$ outputs a valid proof, the pair output by $\mathcal{B}_{\mathsf{PKE}}$ satisfies $e(\pi_V^{(1)}, \mathsf{vk}_\alpha) = e(\mathsf{vk}_1, \pi_V^{(2)})$ (indeed, this is just one of the checks performed by the SNARK verifier $V(\mathsf{vk}, \vec{x}, \pi)$). The d-PKE assumption implies that there is an extractor $\mathcal{E}_{\mathcal{B}_{\mathsf{PKE}}}$ that, given the same input as $\mathcal{B}_{\mathsf{PKE}}$ (including $\mathcal{B}_{\mathsf{PKE}}$'s randomness tape), outputs a vector $\vec{c}$ such that $\pi_V^{(1)} = \sum_{i=0}^d c_i \tau^i \cdot \mathcal{P}_1$.

This extraction step is at the core of the augmented adversary $\mathcal{A}'$. More precisely, on input $(\mathsf{pk}, \mathsf{vk})$ the adversary $\mathcal{A}'$ works as follows:

1. Runs adversary $\mathcal{A}$ on $(\mathsf{pk}, \mathsf{vk})$ and receives its output $(\vec{x}, \pi)$.
2. Uses pk and $\mathsf{trap}_{\mathcal{E}}$ to construct the input σ for $\mathcal{B}_{\mathsf{PKE}}$, and uses pk and vk to construct the auxiliary input z.
3. Runs extractor $\mathcal{E}_{\mathcal{B}_{\mathsf{PKE}}}$ on σ, z and the same randomness tape as used in Step 1.
4. When $\mathcal{E}_{\mathcal{B}_{\mathsf{PKE}}}$ returns $\vec{c}$, $\mathcal{A}'$ sets $\delta := c_d$, $v(z) := \sum_{i=0}^d c_i z^i - \delta t(z)$, and returns the quadruple $(\vec{x}, \pi, \delta, v(z))$.

As the polynomial $t(z)$ is monic we have that $\deg v(z) \leq d - 1$ and $\pi_V^{(2)} = \alpha(v(\tau) + \delta t(\tau)) \cdot \mathcal{P}_2$, as required. □

Lemma A.2.8. *Assume that $d(\lambda)$-TSDH assumption holds relative to bilinear group generator $\mathcal{G}$. Then for every security function $\lambda \colon \mathbb{N} \to \mathbb{N}$, every square span program $(\vec{v}, t)$ of degree $d(\lambda)$,* *and every polynomial-size augmented adversary $\mathcal{A}'$ (see Lemma A.2.7) we have the following:*

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk},\vec{x},\pi)=1 & \mathsf{gk}\leftarrow\mathcal{G}(1^\lambda) \\ \textit{and} & (\mathsf{pk},\mathsf{vk},\mathsf{trap}_\mathcal{S},\mathsf{trap}_\mathcal{E})\leftarrow G(\mathsf{gk},(\vec{v},t)) \\ v(z)^2-1 \textit{ is not divisible by } t(z) & (\vec{x},\pi,\delta,v(z))\leftarrow\mathcal{A}'(\mathsf{pk},\mathsf{vk},\mathsf{trap}_\mathcal{E}) \end{array}\right]\leq\mathsf{negl}(\lambda)\ .$$

Proof. Assume that the statement is false and there exists an augmented adversary $\mathcal{A}'$ for which the above probability is non-negligible. Then we can use $\mathcal{A}'$ to break the d-TSDH assumption with non-negligible probability. More precisely, we construct the d-TSDH adversary $\mathcal{B}_{\mathsf{TSDH}}$ that on input $\sigma := (\mathsf{gk},\mathcal{P}_1,\tau\cdot\mathcal{P}_1,\ldots,\tau^d\cdot\mathcal{P}_1,\mathcal{P}_2,\tau\cdot\mathcal{P}_2,\ldots,\tau^d\cdot\mathcal{P}_2)$ works as follows:

1. $\mathcal{B}_{\mathsf{TSDH}}$ samples random $\alpha,\beta,\rho\in\mathbb{F}_r$ and uses those and the elements of σ to construct the keypair $(\mathsf{pk},\mathsf{vk})$ and extraction trapdoor $\mathsf{trap}_\mathcal{E}$ from the same distribution as output by $G(\mathsf{gk},(\vec{v},t))$.
2. $\mathcal{B}_{\mathsf{TSDH}}$ samples random $\alpha,\beta,\rho\in\mathbb{F}_r$ and uses those and the elements of σ to construct the keypair $(\mathsf{pk},\mathsf{vk})$ and extraction trapdoor $\mathsf{trap}_\mathcal{E}$ from the same distribution as output by $G(\mathsf{gk},(\vec{v},t))$. Every element of the keypair is of form $p(\tau)\cdot\mathcal{P}_1$ or $p(\tau)\cdot\mathcal{P}_2$ for some polynomial p whose coefficients depend on α,β,ρ. Moreover, the degree of p is at most d, so $\mathcal{B}_{\mathsf{TSDH}}$ can always find the necessary encodings of τ^i in its challenge. Just like G, $\mathcal{B}_{\mathsf{TSDH}}$ sets $\mathsf{trap}_\mathcal{E}:=\perp$. The values of τ and α,β,ρ are all random, so the pair $((\mathsf{pk},\mathsf{vk}),\mathsf{trap}_\mathcal{E})$ has the same distribution as induced by G.
3. $\mathcal{B}_{\mathsf{TSDH}}$ runs the augmented SSP adversary $\mathcal{A}'$ on $(\mathsf{pk},\mathsf{vk},\mathsf{trap}_\mathcal{E})$. When $\mathcal{A}'$ returns the quadruple $(\vec{x},\pi,\delta,v(z))$, $\mathcal{B}_{\mathsf{TSDH}}$ parses the proof as $\pi=(\pi_\mathsf{H},\pi_w,\pi_{w\beta},\pi_V^{(2)})$.
4. Successful verification implies that $e(\pi_V^{(1)},\pi_V^{(2)})=e(\pi_\mathsf{H},\mathsf{vk}_{\alpha t})\cdot e(\mathsf{vk}_1,\mathsf{vk}_\alpha)$. The polynomial $t(z)$ does not divide $v(z)^2-1$, so $\mathcal{B}_{\mathsf{TSDH}}$ computes a root d of $t(z)$ that is not a root of $v(z)^2-1$ and writes $(v(z)+\delta t(z))^2-1$ as $a(z)(z-d)+b$ where $b\neq 0$. Note that $e(\pi_V^{(1)},\pi_V^{(2)})=e(\mathcal{P}_1,\mathcal{P}_2)^{\alpha(v(\tau)+\delta t(\tau))^2}$, so we have $e(\mathcal{P}_1,\mathcal{P}_2)^{\alpha(a(\tau)(\tau-d)+b)}=e(\pi_\mathsf{H},\mathsf{vk}_{\alpha t})=e(\pi_\mathsf{H},\alpha t(\tau)\cdot\mathcal{P}_2)$, or $e(\mathcal{P}_1,\mathcal{P}_2)^{a(\tau)/b+\frac{1}{\tau-d}}=e(\pi_\mathsf{H},t(\tau)/((\tau-d)b)\cdot\mathcal{P}_2)$.
5. $\mathcal{B}_{\mathsf{TSDH}}$ uses terms of its challenge to compute $T:=e(\mathcal{P}_1,\mathcal{P}_2)^{a(\tau)/b}$ and $\mathcal{Q}=t(\tau)/((\tau-d)b)\cdot\mathcal{P}_2$; it can always do so as $\deg a(z)\leq 2d-1$ and $\deg t(z):=d$. In particular, $\mathcal{B}_{\mathsf{TSDH}}$ can compute powers $e(\mathcal{P}_1,\mathcal{P}_2)^i$ with $i>d$ by evaluating $e(\tau^i\cdot\mathcal{P}_1,\tau^{i-d}\cdot\mathcal{P}_2)$.
6. $\mathcal{B}_{\mathsf{TSDH}}$ computes $Y:=e(\pi_\mathsf{H},\mathcal{Q})\cdot T^{-1}$ and returns the pair (d,Y).

By construction $Y:=e(\mathcal{P}_1,\mathcal{P}_2)^{\frac{1}{\tau-d}}$ so, whenever $\mathcal{A}'$ returns a valid proof with $v(z)^2-1$ not divisible by $t(z)$, $\mathcal{B}_{\mathsf{TSDH}}$ wins the d-TSDH security game. □

Lemma A.2.9. *Assume that $d(\lambda)$-PDH assumption holds relative to bilinear group generator $\mathcal{G}$.*

Then for every security function $\lambda \colon \mathbb{N} \to \mathbb{N}$, every square span program $(\vec{v}, t)$ of degree $d(\lambda)$, and every polynomial-size augmented adversary $\mathcal{A}'$ (see Lemma A.2.7) we have the following:

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk}, \vec{x}, \pi) = 1 & \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda) \\ \textit{and} & (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{S}}, \mathsf{trap}_{\mathcal{E}}) \leftarrow G(\mathsf{gk}, (\vec{v}, t)) \\ v_{\mathsf{mid}}(z) \notin \operatorname{span}\{v_{n+1}(z), \ldots, v_m(z), t(z)\} & (\vec{x}, \pi, \delta, v(z)) \leftarrow \mathcal{A}'(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{E}}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ ,$$

where $v_{\mathsf{mid}}(z) := v(z) + \delta t(z) - v_0(z) - \sum_{i=1}^{n} x_i v_i(z)$.

Proof. Assume that the statement is false and there exists an augmented adversary $\mathcal{A}'$ for which the above probability is non-negligible. Then we can use $\mathcal{A}'$ to break the (d+1)-PDH assumption with non-negligible probability. More precisely, we construct the (d+1)-PDH adversary $\mathcal{B}_{\mathsf{PDH}}$ that on input $\sigma := (\mathsf{gk}, \mathcal{P}_1, \tau \cdot \mathcal{P}_1, \ldots, \tau^{d+1} \cdot \mathcal{P}_1, \tau^{d+3} \cdot \mathcal{P}_1, \ldots, \tau^{2d+2} \cdot \mathcal{P}_1, \mathcal{P}_2, \tau \cdot \mathcal{P}_2, \ldots, \tau^{d+1} \cdot \mathcal{P}_2, \tau^{d+3} \cdot \mathcal{P}_2, \ldots, \tau^{2d+2} \cdot \mathcal{P}_2)$ works as follows:

1. $\mathcal{B}_{\mathsf{PDH}}$ picks a random degree $d+2$ polynomial $a(z)$ such that, for each of the polynomials $a(z) \cdot v_{n+1}(z), \ldots, a(z) \cdot v_m(z)$ and $a(z) \cdot t(z)$ the coefficient of z^{d+2} is 0. By Lemma A.1.1 it can always do so.
2. $\mathcal{B}_{\mathsf{PDH}}$ samples random α and $\beta' \neq 0$ and uses the elements of σ to construct the keypair $(\mathsf{pk}, \mathsf{vk})$ from the same distribution as output by $G(\mathsf{gk}, (\vec{v}, t))$ for $\alpha := \alpha$, $\beta := \beta' a(\tau)$. The terms $\mathsf{pk}_{\mathsf{H}}^{(1)}$, $\mathsf{pk}_{\mathsf{H}}^{(2)}$, $\mathsf{vk}_{\alpha t}$, vk_{α} and $\mathsf{vk}_{\mathsf{IC}}$ do not involve β and can be computed directly from the challenge. The terms $\mathsf{pk}_{\mathsf{K}} := (\beta v_{n+1}(\tau) \cdot \mathcal{P}_1, \ldots, \beta v_m(\tau) \cdot \mathcal{P}_1)$ and $\mathsf{pk}_{\beta t} := \beta t(\tau) \cdot \mathcal{P}_1$ are known linear combinations of challenge elements and, by choice of $a(z)$, do not involve the term $\tau^{d+2} \cdot \mathcal{P}_1$. Finally, $\mathcal{B}_{\mathsf{PDH}}$ samples $\rho' \in \mathbb{F}_r^*$ and constructs elements $\mathsf{vk}_{\rho} = \rho' t(\tau) \mathcal{P}_2$ and $\mathsf{vk}_{\rho\beta} = \rho' \beta t(\tau) \cdot \mathcal{P}_2$. This corresponds to computing $\mathsf{vk}_{\rho} := \rho \cdot \mathcal{P}_2$ and $\mathsf{vk}_{\rho\beta} := \rho\beta \cdot \mathcal{P}_2$ for implicitly defined $\rho = \rho' t(\tau)$. Note β and ρ are uniformly random, as β' and ρ', respectively, are and therefore the keypair constructed above has the same distribution as keypair output by G. Just like G, $\mathcal{B}_{\mathsf{PDH}}$ sets $\mathsf{trap}_{\mathcal{E}} := \perp$.
3. $\mathcal{B}_{\mathsf{PDH}}$ runs the augmented SSP adversary $\mathcal{A}'$ on $(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{E}})$. When $\mathcal{A}'$ returns the quadruple $(\vec{x}, \pi, \delta, v(z))$, $\mathcal{B}_{\mathsf{TSDH}}$ parses the proof as $\pi = (\pi_{\mathsf{H}}, \pi_w, \pi_{w\beta}, \pi_V^{(2)})$.
4. The random choice of β' hides all information about $a(z)$ from the adversary, therefore we can apply Lemma A.1.1: if $v_{\mathsf{mid}}(z)$ is outside the span of $\{v_{n+1}(z), \ldots, v_m(z), t(z)\}$, then $v_{\mathsf{mid}}(z) \cdot a(z)$ has a non-zero coefficient for z^{d+2}. That is, $v_{\mathsf{mid}}(z) \cdot a(z) = \sum_{i=0}^{2d+2} c_i z^i$ with $c_{d+2} \neq 0$. Because the proof passes the verifiers tests, we have that $\pi_w = \pi_V^{(1)} - (\mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^{n} x_i \cdot \mathsf{vk}_{\mathsf{IC},i}) = v_{\mathsf{mid}}(\tau) \cdot \mathcal{P}_1$ and $\pi_{w\beta} = \beta \cdot \pi_w = \beta' a(\tau) v_{\mathsf{mid}}(\tau) \cdot \mathcal{P}_1$.

5. $\mathcal{B}_{\mathsf{PDH}}$ uses $\pi_{w\beta}$ and elements of its challenge to compute and return $\tau^{d+2} \cdot \mathcal{P}_1$, as follows:

$$\tau^{d+2} \cdot \mathcal{P}_1 = \frac{1}{\beta' c_{d+2}} \left(\pi_{w\beta} - \sum_{\substack{i=0 \\ i \neq d+2}}^{2d+2} \beta' c_i \tau^i \cdot \mathcal{P}_1 \right) .$$

We conclude that whenever $\mathcal{A}'$ returns an accepting proof with $v_{\mathsf{mid}}(z)$ not in the expected span, $\mathcal{B}_{\mathsf{PDH}}$ correctly answers its challenge. □

Proof of Theorem A.2.6. Assume, without loss of generality, that we have access to augmented adversary $\mathcal{A}'$ (see Lemma A.2.7 for details) and let $(\vec{x}, \pi, \delta, v(z))$ be the quadruple returned by $\mathcal{A}'$. We will show that, whenever the instance-proof pair $(\vec{x}, \pi)$ is accepted by the verifier, we can use δ and $v(z)$ to recover a full satisfying assignment $\vec{a}$ for this instance.

Define $v_{\mathsf{mid}}(z) := v(z) + \delta t(z) - v_0(z) - \sum_{i=1}^{n} x_i v_i(z)$. By Lemma A.2.9, $v_{\mathsf{mid}}(z)$ is in the span of $\{v_{n+1}(z), \ldots, v_m(z), t(z)\}$. Express $v_{\mathsf{mid}}(z)$ as $\mathbb{F}_r$-linear combination of these vectors: $v_{\mathsf{mid}}(z) = \sum_{i=n+1}^{m} a_i v_i(z) + \delta' t(z)$. Then, setting $a_i = x_i$ for $1 \leq i \leq n$, we have: $v(z) = v_0(z) + \sum_{i=1}^{m} a_i v_i(z) + (\delta' - \delta) t(z)$. By Lemma A.2.8, $t(z)$ divides $v(z)^2 - 1$, which means that $t(z)$ also divides $\left(v_0(z) + \sum_{i=1}^{m} a_i v_i(z)\right)^2 - 1$, and therefore $(\vec{x}, \vec{a}) \in \mathcal{R}_{(\vec{v},t)}$. We conclude that there exists the required SSP extractor: $\mathcal{E}_{\mathcal{A}}$ performs the above steps and returns $\vec{a}$. □

Remark A.2.10. Assuming the very mild condition that the number of auxiliary variables is at least one less than the number of constraints, i.e., $d > m - n + 1$, one can prove Lemma A.2.9 by relying on d-PDH assumption (see Remark A.1.2); this is the approach of [DFGK14]. We choose to not place any restrictions on the SSP and thus rely on $(d+1)$-PDH assumption instead.

Remark A.2.11. The terms involving β are the only ones that are language-specific and, in essence, β "guards" against choosing $v(z)$ not in the span of the language-specific polynomials $v_i(z)$. If one revealed the encodings of $\beta\tau^i$, the adversary would be able to arbitrarily change the language.

Remark A.2.12. The only reason to choose random ρ is that we don't know how to generate an encoding of β in the $(d+1)$-PDH reduction, so we generate two values $\rho \cdot \mathcal{P}_2$ and $\rho\beta \cdot \mathcal{P}_2$ instead. However, we could set $\rho := t(\tau)$ without breaking the security proof.

A.2.1 MPC generator in duplex pairing setting

The computation $C(\alpha, \beta, \rho, \tau) \cdot (\mathcal{P}_1, \mathcal{P}_2)$ computes the following outputs:

$$\begin{aligned}
\Big(&1, \tau, \ldots, \tau^d, \\
&\alpha, \alpha\tau, \ldots, \alpha\tau^d, \\
&v_0(\tau), \ldots, v_m(\tau), t(\tau), \\
&v_{n+1}(\tau)\beta, \ldots, v_m(\tau)\beta, t(\tau)\beta, \\
&\alpha, \rho, \rho\beta, \alpha t(\tau), v_0(\tau), \ldots, v_n(\tau)\Big) \cdot (\mathcal{P}_1, \mathcal{P}_2) \ .
\end{aligned}$$

Crucially the auxiliary input is independent of α (the terms that involve β, ρ do not involve α), so if we are presented a pair $(x \cdot \mathcal{P}_1, \alpha x \cdot \mathcal{P}_2)$, we can always decompose x as sum of powers of τ.

A.3 The PGHR zk-SNARK protocol

Definition A.3.1. *A* ***quadratic arithmetic program (QAP)*** *of size m and degree d over a field $\mathbb{F}$ is a quadruple $(\vec{A}, \vec{B}, \vec{C}, Z)$, where each of $\vec{A}$, $\vec{B}$ and $\vec{C}$ is a vector of $m+1$ polynomials in $\mathbb{F}^{\leq d-1}[z]$, and $Z(z) \in \mathbb{F}[z]$ is a monic polynomial of degree exactly d.*

Definition A.3.2. *We say that a size-m QAP $(\vec{A}, \vec{B}, \vec{C}, Z)$ is* ***non-degenerate*** *for inputs of size $n \leq m$, if it satisfies the following two conditions: (a) the polynomials $A_0(z), \ldots, A_n(z)$ are all linearly independent; and (b) the spans* $\mathrm{span}\{A_0(z), \ldots, A_n(z)\}$ *and* $\mathrm{span}\{A_{n+1}(z), \ldots, A_m(z)\}$ *are disjoint, except at the origin.*

Definition A.3.3. *The* ***satisfaction problem*** *of a size-m QAP $(\vec{A}, \vec{B}, \vec{C}, Z)$ is the relation $\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$ of pairs $(\vec{x}, \vec{a}) \in \mathbb{F}^n \times \mathbb{F}^m$ satisfying the following conditions: (a) $n \leq m$ and $x_i = a_i$ for $1 \leq i \leq n$ (that is, $\vec{a}$ extends $\vec{x}$); and (b) $Z(z)$ divides the polynomial $(A_0(z) + \sum_{i=1}^m a_i A_i(z)) \cdot (B_0(z) + \sum_{i=1}^m a_i B_i(z)) - (C_0(z) + \sum_{i=1}^m a_i C_i(z))$.*

For the purposes of completeness and to fix notation, in Figure A.2 below we recall the zk-SNARK protocol of Parno et al. [PGHR13]. The zk-SNARK can be used to prove/verify satisfiability of $\mathbb{F}_r$-arithmetic circuits, where r is the order of the two cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$, forming the domain of the pairing $e \colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.

Definition A.3.4. *We define three classes of auxiliary input generators, $\mathcal{Z}_{\mathsf{QAP,A}}$, $\mathcal{Z}_{\mathsf{QAP,B}}$ and $\mathcal{Z}_{\mathsf{QAP,C}}$, as follows.*

- *On input* $(\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \tau)$ *an auxiliary input generator* $Z^{\mathsf{A}}_{(\vec{A},\vec{B},\vec{C},Z)} \in \mathcal{Z}_{\mathsf{QAP,A}}$ *samples random* $\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \beta, \gamma \in \mathbb{F}_r$ *and returns the tuple*

$$\{\rho'_{\mathsf{A}}\} \cup \left(\begin{array}{ccc} \multicolumn{3}{c}{\perp} \\ \alpha_{\mathsf{B}} \cdot \mathcal{P}_1 & \{B_i(\tau)\rho_{\mathsf{B}} \cdot \mathcal{P}_2\}_{i=0}^{m+3} & \{B_i(\tau)\alpha_{\mathsf{B}}\rho_{\mathsf{B}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} \\ \alpha_{\mathsf{C}} \cdot \mathcal{P}_2 & \{C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} & \{C_i(\tau)\alpha_{\mathsf{C}}\rho_{\mathsf{A}}\rho_{\mathsf{B}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} \\ \multicolumn{3}{c}{\{\beta(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}) \cdot \mathcal{P}_1\}_{i=0}^{m+3}} \end{array}\right) \cup \left(\begin{array}{c} \gamma \cdot \mathcal{P}_2 \\ \gamma\beta \cdot \mathcal{P}_1 \\ \gamma\beta \cdot \mathcal{P}_2 \\ Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}} \cdot \mathcal{P}_2 \end{array}\right) .$$

- *On input* $(\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \tau)$ *an auxiliary input generator* $Z^{\mathsf{B}}_{(\vec{A},\vec{B},\vec{C},Z)} \in \mathcal{Z}_{\mathsf{QAP,B}}$ *samples random* $\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{C}}, \beta, \gamma \in \mathbb{F}_r$ *and returns the tuple*

$$\{\rho'_{\mathsf{B}}\} \cup \left(\begin{array}{ccc} \alpha_{\mathsf{A}} \cdot \mathcal{P}_2 & \{A_i(\tau)\rho_{\mathsf{A}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} & \{A_i(\tau)\alpha_{\mathsf{A}}\rho_{\mathsf{A}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} \\ \multicolumn{3}{c}{\perp} \\ \alpha_{\mathsf{C}} \cdot \mathcal{P}_2 & \{C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} & \{C_i(\tau)\alpha_{\mathsf{C}}\rho_{\mathsf{A}}\rho_{\mathsf{B}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} \\ \multicolumn{3}{c}{\{\beta(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}) \cdot \mathcal{P}_1\}_{i=0}^{m+3}} \end{array}\right) \cup \left(\begin{array}{c} \gamma \cdot \mathcal{P}_2 \\ \gamma\beta \cdot \mathcal{P}_1 \\ \gamma\beta \cdot \mathcal{P}_2 \\ Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}} \cdot \mathcal{P}_2 \end{array}\right) .$$

- *On input* $(\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \tau)$ *an auxiliary input generator* $Z^{\mathsf{C}}_{(\vec{A},\vec{B},\vec{C},Z)} \in \mathcal{Z}_{\mathsf{QAP,C}}$ *samples random* $\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \beta, \gamma \in \mathbb{F}_r$ *and returns the tuple*

$$\{\rho'_{\mathsf{A}}\rho'_{\mathsf{B}}\} \cup \left(\begin{array}{ccc} \alpha_{\mathsf{A}} \cdot \mathcal{P}_2 & \{A_i(\tau)\rho_{\mathsf{A}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} & \{A_i(\tau)\alpha_{\mathsf{A}}\rho_{\mathsf{A}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} \\ \alpha_{\mathsf{B}} \cdot \mathcal{P}_1 & \{B_i(\tau)\rho_{\mathsf{B}} \cdot \mathcal{P}_2\}_{i=0}^{m+3} & \{B_i(\tau)\alpha_{\mathsf{B}}\rho_{\mathsf{B}} \cdot \mathcal{P}_1\}_{i=0}^{m+3} \\ \multicolumn{3}{c}{\perp} \\ \multicolumn{3}{c}{\{\beta(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}) \cdot \mathcal{P}_1\}_{i=0}^{m+3}} \end{array}\right) \cup \left(\begin{array}{c} \gamma \cdot \mathcal{P}_2 \\ \gamma\beta \cdot \mathcal{P}_1 \\ \gamma\beta \cdot \mathcal{P}_2 \\ \perp \end{array}\right) .$$

Everywhere above we take ρ_{A} *and* ρ_{B} *to have the following implicit values:* $\rho_{\mathsf{A}} := \rho'_{\mathsf{A}}\tau^{d+1}$ *and* $\rho_{\mathsf{B}} := \rho'_{\mathsf{B}}\tau^{2(d+1)}$, *and just like the generator* G *we set* $A_{m+1} = B_{m+2} = C_{m+3} = Z$ *and* $A_{m+2} = A_{m+3} = B_{m+1} = B_{m+3} = C_{m+1} = C_{m+2} = 0$.

Theorem A.3.5. *Assume that, relative to bilinear group generator* $\mathcal{G}$, *the* $d(\lambda)$*-PDH and* $d(\lambda)$*-TSDH assumptions hold, and* $d(\lambda)$*-PKE assumption holds for the classes* $\mathcal{Z}_{\mathsf{QAP,A}}$, $\mathcal{Z}_{\mathsf{QAP,B}}$ *and* $\mathcal{Z}_{\mathsf{QAP,C}}$. *Then the proof system in Figure A.2 is proof-of-knowledge. That is, for every security function* $\lambda \colon \mathbb{N} \to \mathbb{N}$, *every input size* $n(\lambda)$, *every quadratic arithmetic program* $(\vec{A}, \vec{B}, \vec{C}, Z)$ *of degree* $d(\lambda)$ *that is non-degenerate for inputs of size* $n(\lambda)$, *and every polynomial-size adversary* $\mathcal{A}$, *there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$ such that:*

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk},\vec{x},\pi)=1 & \mathsf{gk}\leftarrow\mathcal{G}(1^\lambda) \\ \textit{and} & (\mathsf{pk},\mathsf{vk},\mathsf{trap}_{\mathcal{S}},\mathsf{trap}_{\mathcal{E}})\leftarrow G(\mathsf{gk},(\vec{A},\vec{B},\vec{C},Z)) \\ (\vec{x},\vec{a})\notin\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)} & ((\vec{x},\pi);\vec{a})\leftarrow(\mathcal{A}(\mathsf{pk},\mathsf{vk})\|\mathcal{E}_{\mathcal{A}}(\mathsf{pk},\mathsf{vk},\mathsf{trap}_{\mathcal{E}})) \end{array}\right]\leq\mathsf{negl}(\lambda)\ .$$

We prove the theorem by introducing and proving the three following lemmas.

Lemma A.3.6. *Assume that, $d(\lambda)$-PKE assumption holds for the for the classes $\mathcal{Z}_{\mathsf{QAP,A}}$, $\mathcal{Z}_{\mathsf{QAP,B}}$ and $\mathcal{Z}_{\mathsf{QAP,C}}$ relative to a bilinear group generator $\mathcal{G}$. Then, for every security function $\lambda\colon\mathbb{N}\to\mathbb{N}$, every input size $n(\lambda)$, every quadratic arithmetic program $(\vec{A},\vec{B},\vec{C},Z)$ of degree $d(\lambda)$ that is non-degenerate for inputs of size $n(\lambda)$, and every polynomial-size adversary $\mathcal{A}$, there exists a polynomial-size adversary $\mathcal{A}'$ such that:*

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk},\vec{x},\pi)=0 & \\ \textit{or all of the following hold:} & \\ \tilde{\vec{x}}=\vec{x},|\vec{x}|=n & \mathsf{gk}\leftarrow\mathcal{G}(1^\lambda) \\ \tilde{\pi}=\pi & (\mathsf{pk},\mathsf{vk},\mathsf{trap}_{\mathcal{S}},\mathsf{trap}_{\mathcal{E}})\leftarrow G(\mathsf{gk},(\vec{A},\vec{B},\vec{C},Z)) \\ \pi'_{\mathsf{A}}=(v'_{\mathsf{A}}(\tau)+\delta_1 Z(\tau))\alpha_{\mathsf{A}}\rho_{\mathsf{A}}\cdot\mathcal{P}_1 & (\vec{x},\pi)\leftarrow\mathcal{A}(\mathsf{pk},\mathsf{vk}) \\ \pi'_{\mathsf{B}}=(v_{\mathsf{B}}(\tau)+\delta_2 Z(\tau))\alpha_{\mathsf{B}}\rho_{\mathsf{B}}\cdot\mathcal{P}_1 & (\tilde{\vec{x}},\tilde{\pi},v'_{\mathsf{A}}(z),v_{\mathsf{B}}(z),v_{\mathsf{C}}(z),\delta_1,\delta_2,\delta_3)\leftarrow\mathcal{A}'(\mathsf{pk},\mathsf{vk},\mathsf{trap}_{\mathcal{E}}) \\ \pi'_{\mathsf{C}}=(v_{\mathsf{C}}(\tau)+\delta_3 Z(\tau))\alpha_{\mathsf{C}}\rho_{\mathsf{A}}\rho_{\mathsf{B}}\cdot\mathcal{P}_1 & \\ \deg v_{\mathsf{A}}(z),\deg v_{\mathsf{B}}(z),\deg v_{\mathsf{C}}(z)\leq d-1 & \end{array}\right]\geq 1-\mathsf{negl}(\lambda)\ ,$$

where α_{A}, α_{B}, α_{C}, τ and $\mathcal{P}_1$ above refer to the elements sampled by G, and the proof π is interpreted as $(\pi_{\mathsf{A}},\pi'_{\mathsf{A}},\pi_{\mathsf{B}},\pi'_{\mathsf{B}},\pi_{\mathsf{C}},\pi'_{\mathsf{C}},\pi_{\mathsf{K}},\pi_{\mathsf{H}})$.

In other words, whenever $\mathcal{A}$ outputs an accepting instance-proof pair, the adversary $\mathcal{A}'$ also returns the same instance-proof pair together with an "explanation" of the terms π'_{A}, π'_{B}, π'_{C}. We use the term "augmented adversary" to denote adversaries $\mathcal{A}'$ which produce such augmented output.

Proof. The proof is analogous to that of Lemma A.2.7. To obtain the polynomial $v'_{\mathsf{A}}(z)$ and the randomization term δ_1, we use the QAP adversary $\mathcal{A}$ to construct an adversary $\mathcal{B}_{\mathsf{PKE}}$ for the d-PKE assumption. Namely, the adversary $\mathcal{B}_{\mathsf{PKE}}$:

1. uses its challenge $(\mathsf{gk},\mathcal{P}_1,\tau\cdot\mathcal{P}_1,\ldots,\tau^d\cdot\mathcal{P}_1,\mathcal{P}_2,\tau\cdot\mathcal{P}_2,\ldots,\tau^d\cdot\mathcal{P}_2,\alpha_{\mathsf{A}}\cdot\mathcal{P}_1,\alpha_{\mathsf{A}}\tau\cdot\mathcal{P}_1,\ldots,$ $\alpha_{\mathsf{A}}\tau^d\cdot\mathcal{P}_1,\alpha_{\mathsf{A}}\cdot\mathcal{P}_2,\alpha_{\mathsf{A}}\tau\cdot\mathcal{P}_2,\ldots,\alpha_{\mathsf{A}}\tau^d\cdot\mathcal{P}_2)$ and auxiliary input to construct QAP keypair $(\mathsf{pk},\mathsf{vk})$;
2. runs $\mathcal{A}$ on this keypair;
3. when $\mathcal{A}$ returns the proof π, the adversary $\mathcal{B}_{\mathsf{PKE}}$ returns the pair $(\pi_{\mathsf{A}},\pi'_{\mathsf{A}})$.

Whenever the proof returned by $\mathcal{A}$ verifies, the adversary $\mathcal{B}_{\mathsf{PKE}}$ correctly responds to its challenge. □

Lemma A.3.7. *Assume that $(3d(\lambda)+2)$-TSDH assumption holds relative to bilinear group generator $\mathcal{G}$. Then for every security function $\lambda\colon \mathbb{N} \to \mathbb{N}$, every quadratic arithmetic program $(\vec{A}, \vec{B}, \vec{C}, Z)$ of degree $d(\lambda)$, and every polynomial-size augmented adversary $\mathcal{A}'$ (see Lemma A.3.6) we have the following:*

$$\Pr\left[\begin{array}{c|c} V(\mathsf{vk}, \vec{x}, \pi) = 1 & \\ \textit{and} & \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda) \\ v_\mathsf{A}(z)\cdot v_\mathsf{B}(z) - v_\mathsf{C}(z) & (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_\mathcal{S}, \mathsf{trap}_\mathcal{E}) \leftarrow G(\mathsf{gk}, (\vec{A}, \vec{B}, \vec{C}, Z)) \\ \textit{is not divisible by } Z(z) & (\vec{x}, \pi, v'_\mathsf{A}(z), v_\mathsf{B}(z), v_\mathsf{C}(z), \delta_1, \delta_2, \delta_3) \leftarrow \mathcal{A}'(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_\mathcal{E}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ ,$$

where $v_\mathsf{A}(z) := A_0(z) + \sum_{i=1}^{n} x_i A_i(z) + v'_\mathsf{A}(z)$.

Proof. Assume that the statement is false and there exists an augmented adversary $\mathcal{A}'$ for which the above probability is non-negligible. Then we can use $\mathcal{A}'$ to break the $(3d+2)$-TSDH assumption with non-negligible probability. More precisely, we construct the $(3d+2)$-TSDH adversary $\mathcal{B}_\mathsf{TSDH}$ that on input $\sigma := (\mathsf{gk}, \mathcal{P}_1, \tau\cdot\mathcal{P}_1, \ldots, \tau^{3d+2}\cdot\mathcal{P}_1, \mathcal{P}_2, \tau\cdot\mathcal{P}_2, \ldots, \tau^{3d+2}\cdot\mathcal{P}_2)$ works as follows:

1. $\mathcal{B}_\mathsf{TSDH}$ samples random $\rho'_\mathsf{A}, \rho'_\mathsf{B}, \alpha_\mathsf{A}, \alpha_\mathsf{B}, \alpha_\mathsf{C}, \beta, \gamma \in \mathbb{F}_r$ and uses those and the elements of σ to construct the keypair $(\mathsf{pk}, \mathsf{vk})$ and extraction trapdoor $\mathsf{trap}_\mathcal{E}$ from the same distribution as output by $G(\mathsf{gk}, (\vec{A}, \vec{B}, \vec{C}, Z))$. Every element of the keypair is of form $p(\tau)\cdot\mathcal{P}_1$ or $p(\tau)\cdot\mathcal{P}_2$ for some polynomial p whose coefficients depend on $\rho'_\mathsf{A}, \rho'_\mathsf{B}, \alpha_\mathsf{A}, \alpha_\mathsf{B}, \alpha_\mathsf{C}, \beta, \gamma$. Moreover, the degree of p is at most $3d+2$, so $\mathcal{B}_\mathsf{TSDH}$ can always find the necessary encodings of τ^i in its challenge. Just like G, $\mathcal{B}_\mathsf{TSDH}$ sets $\mathsf{trap}_\mathcal{E} := (\rho'_\mathsf{A}, \rho'_\mathsf{B})$. The values of τ and $\rho'_\mathsf{A}, \rho'_\mathsf{B}, \alpha_\mathsf{A}, \alpha_\mathsf{B}, \alpha_\mathsf{C}, \beta, \gamma$ are all random, so the pair $((\mathsf{pk}, \mathsf{vk}), \mathsf{trap}_\mathcal{E})$ has the same distribution as induced by G.
2. $\mathcal{B}_\mathsf{TSDH}$ runs the augmented QAP adversary $\mathcal{A}'$ on $(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_\mathcal{E})$. When $\mathcal{A}'$ returns the tuple $(\vec{x}, \pi, v'_\mathsf{A}(z), v_\mathsf{B}(z), v_\mathsf{C}(z), \delta_1, \delta_2, \delta_3)$, $\mathcal{B}_\mathsf{TSDH}$ parses the proof as $\pi = (\pi_\mathsf{A}, \pi'_\mathsf{A}, \pi_\mathsf{B}, \pi'_\mathsf{B}, \pi_\mathsf{C}, \pi'_\mathsf{C}, \pi_\mathsf{K}, \pi_\mathsf{H})$.
3. Successful verification implies that $e(\mathsf{vk}_{\vec{x}} + \pi_\mathsf{A}, \pi_\mathsf{B}) = e(\pi_\mathsf{H}, \mathsf{vk}_\mathsf{Z}) \cdot e(\pi_\mathsf{C}, \mathsf{vk}_1)$. The polynomial $Z(z)$ does not divide $v_\mathsf{A}(z)\cdot v_\mathsf{B}(z) - v_\mathsf{C}(z)$, and thus doesn't divide $q(z) := (v_\mathsf{A}(z) + \delta_1 Z(z))\cdot(v_\mathsf{B}(z) + \delta_2 Z(z)) - (v_\mathsf{C}(z) + \delta_3 Z(z))$ either. $\mathcal{B}_\mathsf{TSDH}$ computes a root d of $Z(z)$ that is not a root of $q(z)$ and writes $q(z)$ as $a(z)(z-d)+b$ where $b \neq 0$.
   Note that $e(\mathsf{vk}_{\vec{x}} + \pi_\mathsf{A}, \pi_\mathsf{B}) \cdot e(\pi_\mathsf{C}, \mathsf{vk}_1)^{-1} = e(\mathcal{P}_1, \mathcal{P}_2)^{\rho_\mathsf{A}\rho_\mathsf{B}(a(\tau)(\tau-d)+b)}$ so we have $e(\mathcal{P}_1, \mathcal{P}_2)^{\rho_\mathsf{A}\rho_\mathsf{B}(a(\tau)(\tau-d)\cdot}$ $e(\pi_\mathsf{H}, \mathsf{vk}_\mathsf{Z}) = e(\pi_\mathsf{H}, Z(\tau)\rho_\mathsf{A}\rho_\mathsf{B}\cdot\mathcal{P}_2)$, or $e(\mathcal{P}_1, \mathcal{P}_2)^{a(\tau)/b+\frac{1}{\tau-d}} = e(\pi_\mathsf{H}, Z(\tau)/((\tau-d)b)\cdot$

$\mathcal{P}_2)$.

4. $\mathcal{B}_{\mathsf{TSDH}}$ uses terms of its challenge to compute $T := e(\mathcal{P}_1, \mathcal{P}_2)^{a(\tau)/b}$ and $\mathcal{Q} := Z(\tau)/((\tau - d)b) \cdot \mathcal{P}_2$; it can always do so as $\deg a(z) \le 2d - 1$ and $\deg Z(z) = d$.
5. $\mathcal{B}_{\mathsf{TSDH}}$ computes $Y := e(\pi_{\mathsf{H}}, \mathcal{Q}) \cdot T^{-1}$ and returns the pair (d, Y).

By construction $Y = e(\mathcal{P}_1, \mathcal{P}_2)^{\frac{1}{\tau - d}}$ so, whenever $\mathcal{A}'$ returns a valid proof with $v_{\mathsf{A}}(z) \cdot v_{\mathsf{B}}(z) - v_{\mathsf{C}}(z)$ not divisible by $Z(z)$, $\mathcal{B}_{\mathsf{TSDH}}$ wins the $(3d + 2)$-TSDH security game. □

Lemma A.3.8. *Assume that $d(\lambda)$-PDH assumption holds relative to bilinear group generator $\mathcal{G}$. Then for every security function $\lambda \colon \mathbb{N} \to \mathbb{N}$, every quadratic arithmetic program $(\vec{A}, \vec{B}, \vec{C}, Z)$ of degree $d(\lambda)$, and every polynomial-size augmented adversary $\mathcal{A}'$ (see Lemma A.3.6) we have the following:*

$$\Pr\left[\begin{array}{c|c} |\vec{x}| = n, & \\ V(\mathsf{vk}, \vec{x}, \pi) = 1, & \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda) \\ \textit{and} & (\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathsf{S}}, \mathsf{trap}_{\mathcal{E}}) \leftarrow G(\mathsf{gk}, (\vec{A}, \vec{B}, \vec{C}, Z)) \\ r(z) \notin \mathrm{span}(R) & (\vec{x}, \pi, v'_{\mathsf{A}}(z), v_{\mathsf{B}}(z), v_{\mathsf{C}}(z), \delta_1, \delta_2, \delta_3) \leftarrow \mathcal{A}'(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{E}}) \end{array}\right] \le \mathsf{negl}(\lambda) ,$$

where

$$r(z) := \rho'_{\mathsf{A}} z^{d+1}(v_{\mathsf{A}}(z) + \delta_1 Z(z)) + \rho'_{\mathsf{B}} z^{2(d+1)}(v_{\mathsf{B}}(z) + \delta_2 Z(z)) + \rho'_{\mathsf{A}}\rho'_{\mathsf{B}} z^{3(d+1)}(v_{\mathsf{C}}(z) + \delta_3 Z(z)) ,$$

$$v_{\mathsf{A}}(z) := A_0(z) + \sum_{i=1}^{n} x_i A_i(z) + v'_{\mathsf{A}}(z) ,$$

$$R := \{\rho'_{\mathsf{A}} z^{d+1} A_i(z) + \rho'_{\mathsf{B}} z^{2(d+1)} B_i(z) + \rho'_{\mathsf{A}}\rho'_{\mathsf{B}} z^{3(d+1)} C_i(z)\}_{i=0}^{m} \cup \left(\begin{array}{c} \rho'_{\mathsf{A}} z^{d+1} Z(z) \\ \rho'_{\mathsf{B}} z^{2(d+1)} Z(z) \\ \rho'_{\mathsf{A}}\rho'_{\mathsf{B}} z^{3(d+1)} Z(z) \end{array}\right) ,$$

and $\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}$ refer to the elements sampled by G.

Proof. Assume that the statement is false and there exists an augmented adversary $\mathcal{A}'$ for which the above probability is non-negligible. Then we can use $\mathcal{A}'$ to break the $(4d + 4)$-PDH assumption with non-negligible probability. More precisely, we construct the $(4d + 4)$-PDH adversary $\mathcal{B}_{\mathsf{PDH}}$ that on input $\sigma := (\mathsf{gk}, \mathcal{P}_1, \tau \cdot \mathcal{P}_1, \ldots, \tau^{4d+4} \cdot \mathcal{P}_1, \tau^{4d+6} \cdot \mathcal{P}_1, \ldots, \tau^{8d+8} \cdot \mathcal{P}_1, \mathcal{P}_2, \tau \cdot \mathcal{P}_2, \ldots, \tau^{4d+4} \cdot \mathcal{P}_2, \tau^{4d+6} \cdot \mathcal{P}_2, \ldots, \tau^{8d+8} \cdot \mathcal{P}_2)$ works as follows:

1. $\mathcal{B}_{\mathsf{PDH}}$ picks a random degree $4d + 5$ polynomial $a(z)$ such that, for each of the polynomials $p(z) \in R$ the coefficient of z^{4d+5} in $a(z)p(z)$ is 0. Each such $p(z)$ is of degree at most $4d + 3$, so by Lemma A.1.1 $\mathcal{B}_{\mathsf{PDH}}$ can always select such $a(z)$.

2. $\mathcal{B}_{\mathsf{PDH}}$ samples random $\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \beta' \in \mathbb{F}_r$ and uses the elements of σ to construct the keypair $(\mathsf{pk}, \mathsf{vk})$ from the same distribution as output by $G(\mathsf{gk}, (\vec{A}, \vec{B}, \vec{C}, Z))$ for $\beta := \beta' a(\tau)$. The terms $\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}'_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{pk}'_{\mathsf{B}}, \mathsf{pk}_{\mathsf{C}}, \mathsf{pk}'_{\mathsf{C}}, \mathsf{pk}_{\mathsf{H}}, \mathsf{vk}_{\mathsf{A}}, \mathsf{vk}_{\mathsf{B}}, \mathsf{vk}_{\mathsf{C}}, \mathsf{vk}_{\mathsf{Z}}$ and $\mathsf{vk}_{\mathsf{IC}}$ can be computed directly from the challenge, as they only require encodings of τ^i for $i \leq 4d+3$. The terms $\mathsf{pk}_{\mathsf{K},i} := \beta\big(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\big)\mathcal{P}_1$ are known linear combinations of challenge elements and, by choice of $a(z)$, do not involve the term $\tau^{d+5} \cdot \mathcal{P}_1$. Finally, $\mathcal{B}_{\mathsf{PDH}}$ samples $\gamma' \in \mathbb{F}_r^*$, sets $\gamma := \gamma' \rho'_{\mathsf{A}} z^{d+1} Z(z)$ and computes $\mathsf{vk}_\gamma := \gamma \mathcal{P}_2$, $\mathsf{vk}^1_{\beta\gamma} := \gamma\beta\mathcal{P}_1$ and $\mathsf{vk}^2_{\beta\gamma} := \gamma\beta\mathcal{P}_2$. By the choice of γ, this computation doesn't require elements $\tau^{4d+5} \cdot \mathcal{P}_1$ and $\tau^{4d+5} \cdot \mathcal{P}_2$, which the challenge doesn't include. Note that β and γ are uniformly random, as β' and γ', respectively, are and therefore the keypair constructed above has the same distribution as keypair output by G. Just like G, $\mathcal{B}_{\mathsf{PDH}}$ sets $\mathsf{trap}_{\mathcal{E}} := (\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}})$.
3. $\mathcal{B}_{\mathsf{PDH}}$ runs the augmented QAP adversary $\mathcal{A}'$ on $(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{E}})$. When $\mathcal{A}'$ returns the eight-tuple $(\vec{x}, \pi, v'_{\mathsf{A}}(z), v_{\mathsf{B}}(z), v_{\mathsf{C}}(z), \delta_1, \delta_2, \delta_3)$, $\mathcal{B}_{\mathsf{PDH}}$ parses the proof as $\pi = (\pi_{\mathsf{A}}, \pi'_{\mathsf{A}}, \pi_{\mathsf{B}}, \pi'_{\mathsf{B}}, \pi_{\mathsf{C}}, \pi'_{\mathsf{C}}, \pi_{\mathsf{K}}, \pi_{\mathsf{H}})$.
4. The random choice of β' hides all information about $a(z)$ from the adversary, therefore we can apply Lemma A.1.1: if $r(z)$ is outside $\mathrm{span}(R)$, then $r(z) \cdot a(z)$ has a non-zero coefficient for z^{4d+5}. That is, $r(z) \cdot a(z) = \sum_{i=0}^{8d+8} c_i z^i$ with $c_{4d+5} \neq 0$. Because the proof passes the verifiers tests, we have that $\gamma \cdot \pi_{\mathsf{K}} = \gamma\beta(\rho'_{\mathsf{A}}\tau^{d+1}(v_{\mathsf{A}}(\tau) + \delta_1 Z(\tau)) + \rho'_{\mathsf{B}}\tau^{2(d+1)}(v_{\mathsf{B}}(\tau) + \delta_2 Z(\tau)) + \rho'_{\mathsf{A}}\rho'_{\mathsf{B}}\tau^{3(d+1)}(v_{\mathsf{C}}(\tau) + \delta_3 Z(\tau))) \cdot \mathcal{P}_1$, i.e. $\pi_{\mathsf{K}} = \beta' a(\tau) r(\tau) \cdot \mathcal{P}_1$.
5. $\mathcal{B}_{\mathsf{PDH}}$ uses π_{K} and elements of its challenge to compute and return $\tau^{4d+5} \cdot \mathcal{P}_1$, as follows:

$$\tau^{4d+5} \cdot \mathcal{P}_1 = \frac{1}{\beta' c_{4d+5}} \left(\pi_{\mathsf{K}} - \sum_{\substack{i=0 \\ i \neq 4d+5}}^{8d+8} \beta' c_i \tau^i \cdot \mathcal{P}_1 \right) .$$

We conclude that whenever $\mathcal{A}'$ returns an accepting proof with $r(z)$ not in the expected span, $\mathcal{B}_{\mathsf{PDH}}$ correctly answers its challenge. □

Proof of Theorem A.3.5. Assume, without loss of generality, that we have access to augmented adversary $\mathcal{A}'$ (see Lemma A.3.6 for details) and let $(\vec{x}, \pi, v'_{\mathsf{A}}(z), v_{\mathsf{B}}(z), v_{\mathsf{C}}(z), \delta_1, \delta_2, \delta_3)$ be the tuple returned by $\mathcal{A}'$. We will show that, whenever the instance-proof pair $(\vec{x}, \pi)$ is accepted by the verifier, we can use the polynomials $v'_{\mathsf{A}}(z)$, $v_{\mathsf{B}}(z)$, $v_{\mathsf{C}}(z)$ and randomization terms δ_1, δ_2, δ_3 to recover a full satisfying assignment $\vec{a}$ for this instance.

Define polynomials $v_{\mathsf{A}}(z)$ and $r(z)$ as in Lemma A.3.8, that is, $v_{\mathsf{A}}(z) := A_0(z) +$

$\sum_{i=1}^{n} x_i A_i(z) + v'_{\mathsf{A}}(z)$ and $r(z) := \rho'_{\mathsf{A}} z^{d+1}(v_{\mathsf{A}}(z) + \delta_1 Z(z)) + \rho'_{\mathsf{B}} z^{2(d+1)}(v_{\mathsf{B}}(z) + \delta_2 Z(z)) + \rho'_{\mathsf{A}}\rho'_{\mathsf{B}} z^{3(d+1)}(v_{\mathsf{C}}(z) + \delta_3 Z(z))$. By Lemma A.3.8 we know that $r(z)$ is in the span of a certain set of $m+4$ polynomials, and therefore there exist $m+4$ field elements $a_0, \dots, a_m, \delta'_1, \delta'_2, \delta'_3 \in \mathbb{F}_r$ such that:

$$r(z) = \sum_{i=0}^{m} a_i \left(\rho'_{\mathsf{A}} z^{d+1} A_i(z) + \rho'_{\mathsf{B}} z^{2(d+1)} B_i(z) + \rho'_{\mathsf{A}}\rho'_{\mathsf{B}} z^{3(d+1)} C_i(z)\right)$$
$$+ \delta'_1 \rho'_{\mathsf{A}} z^{d+1} Z(z) + \delta'_2 \rho'_{\mathsf{B}} z^{2(d+1)} Z(z) + \delta'_3 \rho'_{\mathsf{A}}\rho'_{\mathsf{B}} z^{3(d+1)} Z(z) \ .$$

We claim that this linear combination is necessarily tied to the QAP instance, and we have $a_0 = 1$ and $a_i = x_i$ for $1 \leq i \leq n$. To see that, we restrict our attention to the contribution monomials of degrees $d+1, \dots, 2d+1$ make to the polynomial $r(z)$ and equate the coefficients. By the definition of $r(z)$ we have $r(z)[d+1,\dots,2d+1] = \rho'_{\mathsf{A}} z^{d+1}(v_{\mathsf{A}}(z) + \delta_1 Z(z)) = \rho'_{\mathsf{A}} z^{d+1}(A_0(z) + \sum_{i=1}^{n} x_i A_i(z) + v'_{\mathsf{A}}(z) + \delta_1 Z(z))$, as all other terms contribute monomials of degree at least $2d+1$. If we apply the same reasoning to the span expression above we obtain $r(z)[d+1,\dots,2d+1] = \rho'_{\mathsf{A}} z^{d+1}(\sum_{i=0}^{m} a_i A_i(z) + \delta'_1 Z(z))$. Moreover, because the monomial z^{2d+1} arises only from the highest degree term of $z^{d+1}Z(z)$ we get that $\delta_1 = \delta'_1$, and therefore $A_0(z) + \sum_{i=1}^{n} x_i A_i(z) + v'_{\mathsf{A}}(z) = \sum_{i=0}^{m} a_i A_i(z)$. Now, because the QAP is non-degenerate for inputs of size n, and the spans $\operatorname{span}\{A_0(z), \dots, A_n(z)\}$ and $\operatorname{span}\{A_{n+1}(z), \dots, A_m(z)\}$ are disjoint, except at the origin. Therefore, if we further restrict $r(z)[d+1,\dots,2d+1]$ to the subspace spanned by $\{z^{d+1}A_0(z), \dots, z^{d+1}A_n(z)\}$, all terms vanish except for $z^{d+1}A_0(z), \dots, z^{d+1}A_i(z)$, and we have $A_0(z) + \sum_{i=1}^{n} x_i A_i(z) = \sum_{i=0}^{n} a_i A_i(z)$. Finally, the non-degeneracy of QAP also means that polynomials $A_0(z), \dots, A_n(z)$ are all linearly independent, and therefore $a_0 = 1$ and $a_i = x_i$ for $1 \leq i \leq n$, as claimed.

By Lemma A.3.7, the polynomial $Z(z)$ divides $v_{\mathsf{A}}(z) \cdot v_{\mathsf{B}}(z) - v_{\mathsf{C}}(z)$. Moreover, by equating the coefficients for $z^{2d+2}, \dots, z^{3d+2}$ and $z^{3d+3}, \dots, z^{4d+3}$, respectively, we obtain that $v_{\mathsf{B}}(z) = B_0(z) + \sum_{i=1}^{m} a_i B_i(z)$, and $v_{\mathsf{C}}(z) = C_0(z) + \sum_{i=1}^{m} a_i C_i(z)$. Therefore $(\vec{x}, (a_1, \dots, a_m)) \in \mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, as the assignment $(a_1, \dots, a_m)$ both extends $\vec{x}$, and also satisfies the QAP divisibility property. We conclude that there exists the required QAP extractor: $\mathcal{E}_{\mathcal{A}}$ performs the above steps and returns $\vec{a}$. □

Remark A.3.9. The only reason to choose random γ is that we don't know how to generate an encoding of β in the $(4d+4)$-PDH reduction, so we generate three values $\gamma \cdot \mathcal{P}_1$ and $\gamma\beta \cdot \mathcal{P}_1$, $\gamma\beta \cdot \mathcal{P}_2$ instead. However, we could set $\gamma := \rho'_{\mathsf{A}} \tau^{d+1} Z(\tau)$ without breaking the security proof. This is similar to how ρ is chosen in the DFGK proof system (see Remark A.2.12).

Remark A.3.10. We choose $\rho_{\mathsf{A}} := \rho'_{\mathsf{A}}\tau^{d+1}$ and $\rho_{\mathsf{B}} := \rho'_{\mathsf{B}}\tau^{d+2}$, so that the polynomial $r(z)$, guaranteed by Lemma A.3.8 to be in the span of $\{\beta(A_i(\tau)\rho'_{\mathsf{A}}\tau^{d+1} + B_i(\tau)\rho'_{\mathsf{B}}\tau^{2(d+1)} + C_i(\tau)\rho'_{\mathsf{A}}\rho'_{\mathsf{B}}\tau^{3(d+1)})\}_i$, has clear domain separation. That is, $\tau^{d+1}, \ldots, \tau^{2d+1}$ correspond to the witness prover used to compute $v_{\mathsf{A}}(z)$; $\tau^{2d+2}, \ldots, \tau^{3d+2}$ correspond to the witness prover used to compute $v_{\mathsf{B}}(z)$; and $\tau^{3d+3}, \ldots, \tau^{4d+3}$ correspond to the witness prover used to compute $v_{\mathsf{C}}(z)$. The proof would still hold if $\rho'_{\mathsf{A}} = \rho'_{\mathsf{B}} = 1$ and would let us set $\mathsf{trap}_{\mathcal{E}} := \perp$. A different choice, $\rho_{\mathsf{A}} = 1$, $\rho_{\mathsf{B}} = \tau^{2(d+1)}$ and $\rho_{\mathsf{C}} = \tau^{d+1}$, would still achieve domain separation, and let us reduce to a weaker PDH assumption. This requires including $\tau^{d+1} \cdot \mathcal{P}_2$ in the verification key and using it in the QAP divisibility check.

Remark A.3.11. Just like in the DFGK proof system (see Remark A.2.11) we can't publish more terms involving β than pk_{K}, lest we risk breaking input consistency. That is, every β term revealed needs to be present in the span guaranteed by Lemma A.3.8. This span, however, requires significant restrictions for the main proof to go through. For example, the non-degeneracy property of Definition A.3.2 directly translates to a span constraint.

Public parameters. A prime r, two cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order r with generators $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, and a pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ (where $\mathbb{G}_T$ is also cyclic of order r).

(a) Key generator G

- INPUTS: quadratic arithmetic program $(\vec{A}, \vec{B}, \vec{C}, Z)$
- OUTPUTS: proving key pk, verification key vk, simulation trapdoor $\mathsf{trap}_{\mathcal{S}}$ and extraction trapdoor $\mathsf{trap}_{\mathcal{E}}$

1. Sample two generators $\mathcal{P}_1 \in \mathbb{G}_1$, $\mathcal{P}_2 \in \mathbb{G}_2$ and seven field elements $\tau, \rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \beta, \gamma \in \mathbb{F}_r$ all at random.
2. Extend $\vec{A}, \vec{B}, \vec{C}$ via
$$A_{m+1} = B_{m+2} = C_{m+3} = Z\,,$$
$$A_{m+2} = A_{m+3} = B_{m+1} = B_{m+3} = C_{m+1} = C_{m+2} = 0\,.$$
3. Set $\rho_{\mathsf{A}} := \rho'_{\mathsf{A}}\tau^{d+1}$, $\rho_{\mathsf{B}} := \rho'_{\mathsf{B}}\tau^{2(d+1)}$.
4. Set $\mathsf{pk} := (\mathsf{gk}, (\vec{A}, \vec{B}, \vec{C}, Z), \mathsf{pk}_{\mathsf{A}}, \mathsf{pk}'_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{pk}'_{\mathsf{B}}, \mathsf{pk}_{\mathsf{C}}, \mathsf{pk}'_{\mathsf{C}}, \mathsf{pk}_{\mathsf{K}}, \mathsf{pk}_{\mathsf{H}})$ where for $i = 0, 1, \ldots, m+3$:
$$\mathsf{pk}_{\mathsf{A},i} := A_i(\tau)\rho_{\mathsf{A}}\mathcal{P}_1\,,\quad \mathsf{pk}'_{\mathsf{A},i} := A_i(\tau)\alpha_{\mathsf{A}}\rho_{\mathsf{A}}\mathcal{P}_1\,,$$
$$\mathsf{pk}_{\mathsf{B},i} := B_i(\tau)\rho_{\mathsf{B}}\mathcal{P}_2\,,\quad \mathsf{pk}'_{\mathsf{B},i} := B_i(\tau)\alpha_{\mathsf{B}}\rho_{\mathsf{B}}\mathcal{P}_1\,,$$
$$\mathsf{pk}_{\mathsf{C},i} := C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\mathcal{P}_1\,, \mathsf{pk}'_{\mathsf{C},i} := C_i(\tau)\alpha_{\mathsf{C}}\rho_{\mathsf{A}}\rho_{\mathsf{B}}\mathcal{P}_1\,,$$
$$\mathsf{pk}_{\mathsf{K},i} := \beta\big(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\big)\mathcal{P}_1\,,$$
and for $i = 0, 1, \ldots, d$, $\mathsf{pk}_{\mathsf{H},i} := \tau^i\mathcal{P}_1$.
5. Set $\mathsf{vk} := (\mathsf{gk}, \mathsf{vk}_1, \mathsf{vk}_{\mathsf{A}}, \mathsf{vk}_{\mathsf{B}}, \mathsf{vk}_{\mathsf{C}}, \mathsf{vk}_{\gamma}, \mathsf{vk}^1_{\beta\gamma}, \mathsf{vk}^2_{\beta\gamma}, \mathsf{vk}_{\mathsf{Z}}, \mathsf{vk}_{\mathsf{IC}})$ where
$$\mathsf{vk}_1 := \mathcal{P}_2\,, \mathsf{vk}_{\mathsf{A}} := \alpha_{\mathsf{A}}\mathcal{P}_2\,, \mathsf{vk}_{\mathsf{B}} := \alpha_{\mathsf{B}}\mathcal{P}_1\,, \mathsf{vk}_{\mathsf{C}} := \alpha_{\mathsf{C}}\mathcal{P}_2$$
$$\mathsf{vk}_{\gamma} := \gamma\mathcal{P}_2\,,\quad \mathsf{vk}^1_{\beta\gamma} := \gamma\beta\mathcal{P}_1\,,\quad \mathsf{vk}^2_{\beta\gamma} := \gamma\beta\mathcal{P}_2\,,$$
$$\mathsf{vk}_{\mathsf{Z}} := Z(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\mathcal{P}_2\,, (\mathsf{vk}_{\mathsf{IC},i})_{i=0}^{n} := (A_i(\tau)\rho_{\mathsf{A}}\mathcal{P}_1)_{i=0}^{n}\,.$$
6. Set $\mathsf{trap}_{\mathcal{S}} := (\mathsf{gk}, \mathcal{P}_1, \mathcal{P}_2, \rho_{\mathsf{A}}, \rho_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \beta, \tau)$.
7. Set $\mathsf{trap}_{\mathcal{E}} := (\rho'_{\mathsf{A}}, \rho'_{\mathsf{B}}) \cup \{(\tau^i, \alpha_{\mathsf{A}}\tau^i, \alpha_{\mathsf{B}}\tau^i, \alpha_{\mathsf{C}}\tau^i) \cdot (\mathcal{P}_1, \mathcal{P}_2)\}_{i=0}^{4d+3}$
8. Output $(\mathsf{pk}, \mathsf{vk}, \mathsf{trap}_{\mathcal{S}}, \mathsf{trap}_{\mathcal{E}})$.

(b) Simulator S

- INPUTS: simulation trapdoor $\mathsf{trap}_{\mathcal{S}}$, input $\vec{x} \in \mathbb{F}_r^n$
- OUTPUTS: simulated proof π

1. Sample $v_{\mathsf{A}}, v_{\mathsf{B}}, v_{\mathsf{C}}, h \in \mathbb{F}_r$ at random.
2. Compute
$$\phi_{\mathsf{A}} := v_{\mathsf{A}} - A_0(\tau) - \textstyle\sum_{i=1}^{n} x_i A_i(\tau)\ ,$$
$$h := (v_{\mathsf{A}}v_{\mathsf{B}} - v_{\mathsf{C}})/Z(\tau)\ ,$$

(c) Prover P

- INPUTS: proving key pk, input $\vec{x} \in \mathbb{F}_r^n$, and assignment $\vec{a} \in \mathbb{F}_r^m$, where $(\vec{x}, \vec{a}) \in \mathcal{R}_{(\vec{A}, \vec{B}, \vec{C}, Z)}$.
- OUTPUTS: proof π

1. Sample $\delta_1, \delta_2, \delta_3 \in \mathbb{F}_r$ at random.
2. Compute $\vec{h} = (h_0, \ldots, h_0) \in \mathbb{F}_r^{d+1}$, the coefficients of the polynomial $H(z) := \frac{v_{\mathsf{A}}(z)v_{\mathsf{B}}(z) - v_{\mathsf{C}}(z)}{Z(z)}$ where $v_{\mathsf{A}}, v_{\mathsf{B}}, v_{\mathsf{C}} \in \mathbb{F}_r[z]$ are as follows:
$$v_{\mathsf{A}}(z) := A_0(z) + \textstyle\sum_{i=1}^{m} a_i A_i(z) + \delta_1 Z(z)\ ,$$
$$v_{\mathsf{B}}(z) := B_0(z) + \textstyle\sum_{i=1}^{m} a_i B_i(z) + \delta_2 Z(z)\ ,$$
$$v_{\mathsf{C}}(z) := C_0(z) + \textstyle\sum_{i=1}^{m} a_i C_i(z) + \delta_3 Z(z)\ .$$
3. Set $\tilde{\mathsf{pk}}_{\mathsf{A}}$:= "same as pk_{A}, but with $\mathsf{pk}_{\mathsf{A},i} = 0$ for $i = 0, 1, \ldots, n$". Set $\tilde{\mathsf{pk}}'_{\mathsf{A}}$:= "same as $\mathsf{pk}'_{\mathsf{A}}$, but with $\mathsf{pk}'_{\mathsf{A},i} = 0$ for $i = 0, 1, \ldots, n$".
4. Letting $\vec{c} := (1 \circ \vec{a} \circ \delta_1 \circ \delta_2 \circ \delta_3) \in \mathbb{F}_r^{4+m}$, compute
$$\pi_{\mathsf{A}} := \langle \vec{c}, \tilde{\mathsf{pk}}_{\mathsf{A}} \rangle,\ \pi'_{\mathsf{A}} := \langle \vec{c}, \tilde{\mathsf{pk}}'_{\mathsf{A}} \rangle,\ \pi_{\mathsf{B}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{B}} \rangle,\ \pi'_{\mathsf{B}} := \langle \vec{c}, \mathsf{pk}'_{\mathsf{B}} \rangle,$$
$$\pi_{\mathsf{C}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{C}} \rangle,\ \pi'_{\mathsf{C}} := \langle \vec{c}, \mathsf{pk}'_{\mathsf{C}} \rangle,\ \pi_{\mathsf{K}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{K}} \rangle,\ \pi_{\mathsf{H}} := \langle \vec{h}, \mathsf{pk}_{\mathsf{H}} \rangle.$$
5. Output $\pi := (\pi_{\mathsf{A}}, \pi'_{\mathsf{A}}, \pi_{\mathsf{B}}, \pi'_{\mathsf{B}}, \pi_{\mathsf{C}}, \pi'_{\mathsf{C}}, \pi_{\mathsf{K}}, \pi_{\mathsf{H}})$.

(d) Verifier V

- INPUTS: verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π
- OUTPUTS: decision bit

1. Compute $\mathsf{vk}_{\vec{x}} := \mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^{n} x_i \cdot \mathsf{vk}_{\mathsf{IC},i} \in \mathbb{G}_1$.
2. Check validity of knowledge commitments for A, B, C:
$$e(\pi_{\mathsf{A}}, \mathsf{vk}_{\mathsf{A}}) = e(\pi'_{\mathsf{A}}, \mathsf{vk}_1)\,, e(\mathsf{vk}_{\mathsf{B}}, \pi_{\mathsf{B}}) = e(\pi'_{\mathsf{B}}, \mathsf{vk}_1)\,,$$
$$e(\pi_{\mathsf{C}}, \mathsf{vk}_{\mathsf{C}}) = e(\pi'_{\mathsf{C}}, \mathsf{vk}_1)\,.$$
3. Check that the same same coefficients were used:
$$e(\pi_{\mathsf{K}}, \mathsf{vk}_{\gamma}) = e(\mathsf{vk}_{\vec{x}} + \pi_{\mathsf{A}} + \pi_{\mathsf{C}}, \mathsf{vk}^2_{\beta\gamma}) \cdot e(\mathsf{vk}^1_{\beta\gamma}, \pi_{\mathsf{B}})\ .$$
4. Check QAP divisibility:
$$e(\mathsf{vk}_{\vec{x}} + \pi_{\mathsf{A}}, \pi_{\mathsf{B}}) = e(\pi_{\mathsf{H}}, \mathsf{vk}_{\mathsf{Z}}) \cdot e(\pi_{\mathsf{C}}, \mathsf{vk}_1)\ .$$
5. If all checks pass, output $b := 1$ (accept), otherwise output $b := 0$ (reject).

Bibliography

[AJL$^+$12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT '12, pages 483–501, 2012.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $o(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, 1983.

[AL11] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. Cryptology ePrint Archive, Report 2011/136, 2011.

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *Proceedings of the 23rd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '04, pages 56–73, 2004.

[BBFR15] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages 271–286, 2015.

[BC89] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *Proceedings of the 9th Annual International Cryptology Conference*, CRYPTO '89, pages 400–407, 1989.

[BC12] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In *Proceedings of the 32nd Annual International Cryptology Conference*, CRYPTO '12, pages 255–272, 2012.

[BCC$^+$14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. ePrint 2014/580, 2014.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, 2012.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of the 45th ACM Symposium on the Theory of Computing*, STOC '13, pages 111–120, 2013.

[BCG$^+$13a] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCG$^+$13b] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. TinyRAM architecture specification v2.00, 2013. URL: `http://scipr-lab.org/tinyram`.

[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 *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[BCG$^+$15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 287–304, 2015.

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th Innovations in Theoretical Computer Science Conference*, ITCS '13, pages 401–414, 2013.

[BCI$^+$13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 315–333, 2013.

[BCKL08] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and noninteractive anonymous credentials. In *Proceedings of the 5th Theory of Cryptography Conference*, TCC '08, pages 356–374, 2008.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. Cryptology ePrint Archive, Report 2014/595, 2014.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the 34th Annual International Cryptology Conference*, CRYPTO '14, pages 276–294, 2014. Extended version at `http://eprint.iacr.org/2014/595`.

[BCTV14c] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 781–796, 2014. Extended version at `http://eprint.iacr.org/2013/879`.

[BDLO12] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In *Proceedings of the 13th International Conference on Cryptology in India*, Indocrypt '12, pages 454–473, 2012.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 257–266, 2008.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In *Proceedings of the 9th International Conference on Security in Communication Networks*, SCN '14, pages 175–196, 2014.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT '11, pages 169–188, 2011.

[BDSMP91] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[BÉ02] Bruno Beauquier and Darrot Éric. On arbitrary size Waksman networks and their vulnerability. *Parallel Processing Letters*, 12(3-4):287–296, 2002.

[Ben65] Václav E. Beneš. *Mathematical theory of connecting networks and telephone traffic*. New York, Academic Press, 1965.

[Ber02] Daniel J. Bernstein. Pippenger's exponentiation algorithm. `http://cr.yp.to/papers/pippenger.pdf`, 2002.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, 1988.

[BFR$^+$13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, 2013.

[BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 390–420, 1993.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, 1988.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 478–492, 2013.

[Bis15] Bryan Bishop. Review of bitcoin scaling proposals. `http://diyhpl.us/~bryan/irc/bitcoin/scalingbitcoin-review.pdf`, 2015.

[BKLS02] Paulo S. L. M. Barreto, Hae Yong Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. In *Proceedings of the 22Nd Annual International Cryptology Conference*, CRYPTO '02, pages 354–368, 2002.

[BLS04] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Efficient implementation of pairing-based cryptosystems. *Journal of Cryptology*, 17(4):321–334, 2004.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography*, SAC'05, pages 319–331, 2006.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, 1988.

[BW06] Xavier Boyen and Brent Waters. Compact group signatures without random oracles. In *Proceedings of the 25th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '06, pages 427–444, 2006.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, FOCS '01, pages 136–145, 2001.

[CFA$^+$12] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2 edition, 2012.

[CFH$^+$15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages 253–270, 2015.

[CH10] Jeremy Clark and Urs Hengartner. On the use of financial data as a random beacon. In *Proceedings of the 2010 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections*, EVT/WOTE '10, 2010.

[Cha82] David Chaum. Blind signatures for untraceable payments. In *Proceedings of the 2nd Annual International Cryptology Conference*, CRYPTO '82, pages 199–203, 1982.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT '05, pages 302–321, 2005.

[CKLM13] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Succinct malleable NIZKs and an application to compact shuffles. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 100–119, 2013.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CP92] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '92, pages 89–105, 1992.

[CPS07] Ran Canetti, Rafael Pass, and Abhi Shelat. Cryptography from sunspots: How to use an imperfect reference string. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 249–259, 2007.

[CS04] John F. Canny and Stephen Sorkin. Practical large-scale distributed key generation. In *Proceedings of the 23rd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '04, pages 138–152, 2004.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Proceedings of the 1st Symposium on Innovations in Computer Science*, ICS '10, pages 310–331, 2010.

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster computing in zero knowledge. In *Proceedings of the 34th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '15, pages 371–403, 2015.

[DFGK14] George Danezis, Cedric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In *Proceedings of the 20th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '14, pages 532–550, 2014.

[DFH12] Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In *Proceedings of the 9th Theory of Cryptography Conference*, TCC '12, pages 54–74, 2012.

[DFKP13] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *Proceedings of the 2013 Workshop on Language Support for Privacy Enhancing Technologies*, PETShop '13, 2013.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography*, PKC '09, pages 160–179, 2009.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - Or: Breaking the SPDZ limits. In *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS '13, pages 1–18, 2013.

[DL08] Giovanni Di Crescenzo and Helger Lipmaa. Succinct NP proofs from an extractability assumption. In *Proceedings of the 4th Conference on Computability in Europe*, CiE '08, pages 175–185, 2008.

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. In *Proceedings of 9th International Conference on Security and Cryptography for Networks*, SCN '14, pages 398–415, 2014.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the 32nd Annual International Cryptology Conference*, CRYPTO '12, pages 643–662, 2012.

[FL14] Matthew Fredrikson and Benjamin Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 909–924, 2014.

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple noninteractive zero knowledge proofs under general assumptions. *SIAM Journal on Computing*, 29(1):1–28, 1999.

[FLZ13] Prastudy Fauzi, Helger Lipmaa, and Bingsheng Zhang. Efficient modular NIZK arguments from shift and product. In *Proceedings of the 12th International Conference on Cryptology and Network Security*, CANS '13, pages 92–121, 2013.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings of the 6th Annual International Cryptology Conference*, CRYPTO '87, pages 186–194, 1987.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010.

[GGHR14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. Two-round secure MPC from indistinguishability obfuscation. In *Proceedings of the 11th Theory of Cryptography Conference*, TCC '14, pages 74–94, 2014.

[GGJS11] Sanjam Garg, Vipul Goyal, Abhishek Jain, and Amit Sahai. Bringing people of different beliefs together to do UC. In *Proceedings of the 8th Theory of Cryptography Conference*, TCC '11, pages 311–328, 2011.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '13, pages 626–645, 2013.

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[GK08] Vipul Goyal and Jonathan Katz. Universally composable multi-party computation with an unreliable common reference string. In *Proceedings of the 5th Theory of Cryptography Conference*, TCC '08, pages 142–154, 2008.

[GLR11] Shafi Goldwasser, Huijia Lin, and Aviad Rubinstein. Delegation of computation without rejection problem from designated verifier CS-proofs. Cryptology ePrint Archive, Report 2011/456, 2011.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. Preliminary version appeared in STOC '85.

[GMW87a] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, 1987.

[GMW87b] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, 1987.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991. Preliminary version appeared in FOCS '86.

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

[GO07] Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. In *Proceedings of the 27th Annual International Cryptology Conference*, CRYPTO '07, pages 323–341, 2007.

[Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *Proceedings of the 46th ACM Symposium on the Theory of Computing*, STOC '14, pages 684–693, 2014.

[GOS06a] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive Zaps and new techniques for NIZK. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, CRYPTO '06, pages 97–111, 2006.

[GOS06b] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for NP. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, EUROCRYPT '06, pages 339–358, 2006.

[Gro05] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *Proceedings of the 3rd International Conference on Applied Cryptography and Network Security*, ACNS '05, pages 467–482, 2005.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *Proceedings of the 12th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT '06, pages 444–459, 2006.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the 35th Annual International Conference on Advances in Cryptology*, EUROCRYPT '16, pages 305–326, 2016.

[GS06] R. Granger and Nigel Smart. On computing products of pairings. Cryptology ePrint Archive, Report 2006/172, 2006.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 99–108, 2011.

[HMRT12] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA key generation and threshold Paillier in the two-party setting. In *Proceedings of the The Cryptographers' Track at the RSA Conference 2012*, CT-RSA 2012, pages 313–331, 2012.

[KAK96] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[KHG12] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. Cryptology ePrint Archive, Report 2012/377, 2012.

[KKZZ14] Jonathan Katz, Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Distributing the setup in universally composable multi-party computation. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 20–29, 2014.

[KMO01] Jonathan Katz, Steven Myers, and Rafail Ostrovsky. Cryptographic counters and applications to electronic voting. In *Proceedings of the 20th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '01, pages 78–92, 2001.

[KOS03] Jonathan Katz, Rafail Ostrovsky, and Adam Smith. Round efficiency of multi-party computation with a dishonest majority. In *Proceedings of the 22Nd International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT '03, pages 578–595. Springer-Verlag, 2003.

[KPP$^+$14] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: Faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 765–780, 2014.

[Lip11] Helger Lipmaa. Two simple code-verification voting protocols. Cryptology ePrint Archive, Report 2011/317, 2011.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*, TCC '12, pages 169–189, 2012.

[Lip13] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *Proceedings of the 19th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '13, pages 41–60, 2013.

[Lip14] Helger Lipmaa. Efficient NIZK arguments via parallel verification of Beneš networks. In *Proceedings of the 9th International Conference on Security and Cryptography for Networks*, SCN '14, pages 416–434, 2014.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22:161–188, April 2009.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[Mie08] Thilo Mie. Polylogarithmic two-round argument systems. *Journal of Mathematical Cryptology*, 2(4):343–363, 2008.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*, SSYM '04, pages 20–20, 2004.

[MNT01] Atsuko Miyaji, Masaki Nakabayashi, and Shunzo Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 84(5):1234–1243, 2001.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[MOV91] Alfred Menezes, Tatsuaki Okamoto, and Scott Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, STOC '91, pages 80–89, 1991.

[MR14] Silvio Micali and Michael O. Rabin. Cryptography miracles, secure auctions, matching problem verification. *Communications of the ACM*, 57(2):85–93, 2014.

[MSKK15] Andrew Miller, Elaine Shi, Ahmed Kosba, and Jonathan Katz. Nonoutsourceable scratch-off puzzles to discourage Bitcoin mining coalitions. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, 2015.

[Nak09] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[Nat12] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard. `http://csrc.nist.gov/publications/PubsFIPS.html`, 2012.

[Nat14] National Institute of Standards and Technology. NIST randomness beacon, 2014. URL: `http://www.nist.gov/itl/csd/ct/nist_beacon.cfm`.

[Orl11] Claudio Orlandi. Is multiparty computation any good in practice? In *Proceedings of the 2011 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '11, pages 5848–5851, 2011.

[Pas04] Rafael Pass. Bounded-concurrent secure multi-party computation with a dishonest majority. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 232–241. ACM, 2004.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '91, pages 129–140, 1992.

[PGHR13] Brian Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP '13, pages 238–252, 2013.

[Pol13] PolarSSL. PolarSSL. `http://polarssl.org`, Oct 2013.

[PRST08] David C. Parkes, Michael O. Rabin, Stuart M. Shieber, and Christopher Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. *Electronic Commerce Research and Applications*, 7(3):294–312, 2008.

[RMMY12] Michael O. Rabin, Yishay Mansour, S. Muthukrishnan, and Moti Yung. Strictly-black-box zero-knowledge and efficient validation of financial transactions. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming*, ICALP '12, pages 738–749, 2012.

[SB04] Michael Scott and Paulo S. L. M. Barreto. Compressed pairings. In *Proceedings of the 24th Annual International Cryptology Conference*, CRYPTO '04, pages 140–156, 2004.

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

[SCI] SCIPR Lab. libsnark: a C++ library for zkSNARK proofs. URL: `https://github.com/scipr-lab/libsnark`.

[Sco05] Michael Scott. Computing the Tate pairing. In *Proceedings of the The Cryptographers' Track at the RSA Conference 2005*, CT-RSA '05, pages 293–304, 2005.

[Sco07] Michael Scott. Implementing cryptographic pairings. In *Proceedings of the 1st First International Conference on Pairing-Based Cryptography*, Pairing '07, pages 177–196, 2007.

[Sil09] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Springer, 2 edition, 2009.

[Sol03] Jerome A. Solinas. Id-based digital signature algorithms. `http://cacr.uwaterloo.ca/conferences/2003/ecc2003/solinas.pdf`, 2003.

[sS13] abhi shelat and Chih-hao Shen. Fast two-party secure computation with minimal assumptions. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 523–534, 2013.

[ST99] Tomas Sander and Amnon Ta-Shma. Auditable, anonymous electronic cash. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 555–572, 1999.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Theory of Cryptography Conference*, TCC '08, pages 1–18, 2008.

[Wak68] Abraham Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[Was08] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC, 2 edition, 2008.

[WSR$^+$15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, NDSS '15, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, 1986.

[ZPK14] Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 856–867, 2014.