Practical and Employable Protocols for UC-Secure Circuit Evaluation over $\mathbb{Z}_{\mathsf{n}}$*

Jan Camenisch[1], Robert R. Enderlein[1,2], and Victor Shoup[3]

[1] IBM Research – Zurich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
[2] Department of Computer Science, ETH Zürich, CH-8092 Zürich, Switzerland
[3] New York University, Courant Institute, NY 10012 New York, United States

Abstract. We present a set of new, efficient, universally composable two-party protocols for evaluating reactive arithmetic circuits modulo n, where n is a safe RSA modulus of unknown factorization. Our protocols are based on a homomorphic encryption scheme with message space $\mathbb{Z}_{\mathsf{n}}$, zero-knowledge proofs of existence, and a novel "mixed" trapdoor commitment scheme. Our protocols are proven secure against *adaptive corruptions* (assuming secure *erasures*) under standard assumptions in the CRS model (without random oracles). Our protocols appear to be the most efficient ones that satisfy these security requirements. In contrast to prior protocols, we provide facilities that allow for the use of our protocols as building blocks of higher-level protocols. An additional contribution of this paper is a universally composable construction of the variant of the Dodis-Yampolskiy oblivious pseudorandom function in a group of order n as originally proposed by Jarecki and Liu.

Keywords: Two-party computation, Practical Protocols, UC-Security.

1 Introduction

Designing and proving secure large and complex cryptographic protocols is very challenging. Today, the security proofs of most practical protocols consider only a single instance of the protocol and therefore all security guarantees are lost if such a protocol is run concurrently with other protocols or with itself, in other words, when used in practice. Better security guarantees can be obtained when using composability frameworks—Canetti's Universal Composability (UC) [Can00], the similar GNUC [HS11] by Hofheinz and Shoup, or other frameworks [PW01,Küs06,MR11]—which ensure that protocols proved secure in the framework remain secure under arbitrary composition. This also simplifies the design of protocols: high-level protocols can be composed from building block protocols and the security proofs of the high-level protocols can be based on the security of the building blocks and so become modular and easier.

Unfortunately, protocols proven secure in such composability frameworks are typically an order of magnitude less efficient than their traditional counterparts with "single-instance" security. Moreover, most UC-secure schemes and protocols found in the literature can not be used as building blocks for higher-level protocols because they do not offer the proper interfaces. That is, unless one considers only multi-party protocols with honest majority, it is typically not possible to ensure that a party's output of one building block is used as the party's input to another building block. We note that the situation for two-party protocols is different from UC-secure multi-party protocols with an honest majority where it is possible to secret-share all input and output values and then, by the virtue of the majority's honesty, it is ensured that the right outputs are used as inputs to the next building block.

* This is the full version of a paper due to appear at the 18th European Symposium on Research in Computer Security (ESORICS 2013). The final publication is available at `link.springer.com`.

In this paper we are therefore interested in practically useful UC-secure building block protocols that provide interfaces so that parties in higher-level protocols can prove to each other that their inputs to one building block protocol correspond to the outputs of another building block protocol. More precisely, we provide a set of two-party protocols for evaluating an arithmetic circuit with reactive inputs and outputs. The protocols accept as (additional) inputs and provide as (additional) outputs tailored commitment values which, in conjunction with UC zero-knowledge proofs, make them a useful building block for higher-level protocols. We demonstrate the usefulness of our protocols by providing as example application an oblivious pseudorandom function evaluation (see Section 8) and point out that our protocols can be used to implement the subprotocols required by Camenisch et al.'s credential authenticated identification and key-exchange protocols [CCGS10] (see Section 6.3 of their paper).

Apart from being the only protocols that allow for their use as building blocks, ours are also more efficient than existing UC-secure two-party reactive circuit evaluation protocols [DN03] [IPS09,DO10a,BDOZ11] which were designed to be used as standalone protocols.

Our contribution. Our main contribution is twofold: *1)* we provide a mechanism for protocol designers to easily integrate our arithmetic circuit functionality in their higher-level protocol in a practical yet secure manner; and *2)* we provide a concrete construction of the circuit evaluation protocol that is in itself more efficient than prior work. We achieve the latter by using cryptographic primitives that work very well together. Additionally, the tools we use in our construction—especially our novel mixed trapdoor commitment scheme—may be of independant interest.

Our protocols evaluate an arithmetic circuit modulo a composite number n, where n is a product of two large safe primes that is assumed to be generated by a trusted third party, and whose factorization remains otherwise unknown. We believe that in many practical cases, this is a natural assumption.

Our protocols are *universally composable* and proven secure under standard assumptions in a setting where parties can be *corrupted at any time*. It additionally assumes that *secure erasures* are possible and that parties can agree on a *common reference string* (CRS). We do not require random oracles. We strongly believe that achieving security against adaptive corruptions is crucial in order to achieve any meaningful sense of security in the "real world", where computers are compromised on a regular basis. The assumption of secure erasures is a pragmatic compromise: without it, obtaining a practical protocol seems unlikely; moreover, this assumption does not seem that unrealistic. Likewise, as it is impossible to achieve universal composability without some kind of setup assumption [CKL06], a CRS seems like a reasonable, pragmatic compromise.

Our ideal functionality. We denote our basic ideal functionality for verifiably evaluating arithmetic circuits modulo n by $\mathcal{F}_{\text{ABB}}$ (our functionality is similar to Nielsen's arithmetic black box [Nie03], hence the name). Parties compute the circuit step-by-step in a reactive manner by sending identical instructions with identical common input to $\mathcal{F}_{\text{ABB}}$. (For some instructions, one party must additionally provide private input to $\mathcal{F}_{\text{ABB}}$.) We assume that a higher-level protocol orchestrates the steps the parties take.

$\mathcal{F}_{\text{ABB}}$ processes instructions from the two parties of the following types: *Input:* a party inserts a value in $\mathbb{Z}_{\mathsf{n}}$ into the circuit; *Linear Combination:* a linear combination of values in the circuit is computed; *Multiplication:* the product of two values in the circuit is computed; *Output:* a

value in the circuit is output to a party; *Proof:* a party can prove an arbitrary statement to the other party in zero-knowledge involving values that she input in the circuit, values she got as an output, and values external to the circuit.

A party can use the *Proof* instruction to prove that the value inside a commitment used in the higher-level protocol is the same as a value in the circuit. This instruction thus makes it easy and practical to compose $\mathcal{F}_{\text{ABB}}$ with a higher-level protocol. To input a committed value from a higher-level protocol into the circuit, $\mathcal{P}$ would first use the *Input* instruction to set the value in the circuit, and then use the *Proof* instruction to convince $\mathcal{Q}$ that the new value corresponds to what was in the commitment. Similarly to transfer a value from the circuit to the higher-level protocol, $\mathcal{P}$ would first get the value with the *Output* instruction, generate a commitment in the higher-level protocol, and then use the *Proof* instruction to convince $\mathcal{Q}$ that the commitment contains the value that was output by the circuit.

All of our results are presented in the GNUC framework [HS11]. This has two advantages. First, the GNUC framework is mathematically consistent, and so our results have a clear mathematical meaning. Second, the GNUC framework supports the notion of a *system parameter*, which is how we wish to model the modulus n (a system parameter is formally modeled as an "ideal functionality", to which all parties—including the environment—have direct access).

Additional features. Our framework can be extended with some features, such as generating random values and computing multiplicative inverses modulo n, using standard techniques. Other features require an extension of our ideal functionality. In Section 5 we add the following instruction to $\mathcal{F}_{\text{ABB}}$: *Exponentiated Output:* given a group of order n, a generator g of that group, and the identifier k of a previously assigned value x, the group element g^x is output to a party. With this feature, we can directly implement Jarecki and Liu's two-party protocol for computing the following oblivious pseudorandom function (OPRF) [JL09]:

$$f_y(x) = \begin{cases} \mathsf{g}^{1/(y+x)} & \text{if } \gcd(y+x, \mathsf{n}) = 1, \\ 1 & \text{otherwise.} \end{cases}$$

Here, $\mathcal{P}$'s private input is x, $\mathcal{Q}$'s private input is y, and $\mathcal{P}$'s output is $f_y(x)$. As pointed out by Jarecki and Liu, OPRF's have many useful cryptographic applications.

Efficiency. Our protocols are quite practical; in particular, they do not require any expensive "cut and choose" techniques. The complexity of our protocols can be summarized as follows: if the circuits involved have t gates, the communication complexity is $O(t)$ elements of $\mathbb{Z}_{\mathsf{n}^2}$ (and groups of similar or smaller order) and the computational complexity is $O(t)$ exponentiations in $\mathbb{Z}_{\mathsf{n}^2}$ (and groups of similar or smaller order). We report on an experimental comparison of our protocols with relevant prior work in Section 7.1. We show that our protocols are practical, and that small circuits can be run in a few seconds—for examplethe OPRF computation above (for a 1248-bit modulus) would run in 0.84 seconds on the authors' laptop computers.

Roadmap. In Section 2 we introduce the notation used in this paper, recapitulate some fundamental theory, and present our new mixed trapdoor commitment scheme. We describe our ideal functionality $\mathcal{F}_{\text{ABB}}$ for circuit evaluation in Section 3, and construct a concrete protocol in Section 4. In Section 5 we add additional features to our functionality. We prove our protocol secure in Section 6. In Section 7 we disucuss related work, and compare the efficiency of our protocols with relevant related work. In Section 8 we show how one can easily construct an OPRF using our protocol.

2 Preliminaries

In this section we will introduce the notation used throughout this paper and provide some background on the UC model, zero-knowledge proofs of existence, and homomorphic encryption. Finally we provide a new construction of a commitment scheme, which might be of independant interest.

2.1 Notation

By $\mathbb{N}_i$ we denote the set of all natural numbers between 0 and $(i-1)$, by $\mathbb{Z}_i$ we denote the ring of integers modulo i. We use $\mathbb{N}_i^*$ and $\mathbb{Z}_i^*$ to denote $\mathbb{N}_i \setminus \{0\}$ and $\mathbb{Z}_i \setminus \{0\}$, respectively. If $\mathbb{A}$ is a set, then $a \xleftarrow{\$} \mathbb{A}$ means we set a to a random element of that set. If A is a Probabilistic Polynomial-Time (PPT) algorithm, then $y \xleftarrow{\$} \mathsf{A}(x)$ means we assign y to the output of $\mathsf{A}(x)$ when run with fresh random coins on input x.

Let Σ denote a fixed, finite alphabet of symbols (for example Unicode codepoints). Throughout this text we will use monospace fonts to denote characters in Σ, e.g.: `P` or `Q`. By $\Sigma^\star$ we denote the set of strings over Σ. We use the list-encoding function $\langle\cdot\rangle$ like in the GNUC paper [HS11]: If $a_1, \ldots, a_n \in \Sigma^\star$, then $\langle a_1, \ldots, a_n\rangle$ is a string over Σ that encodes the list $(a_1, \ldots, a_n)$ in some canonical way.

If AP is a set, $AP \leftarrow k$ is a shorthand notation for inserting k into it: $AP \leftarrow AP \cup k$.

If V is an associative array, then $V[k] \leftarrow v$ denotes the insertion of the value v into the array under the identifier k. By $v' \leftarrow V[k]$, we denote the retrieval of the value associated with identifier k, and storing that retieved value in the variable v'. In this paper, we will never insert the same identifier twice in any array, and we will always use identifiers that were previously input into the array when retrieving a value.

$\mathcal{P}$ and $\mathcal{Q}$ denote the two parties in an interactive protocol, and $\mathcal{A}$ the adversary.

2.2 UC and GNUC Models

Protocols constructed for and proven secure in a composability framework can be securely composed in arbitrary ways. To date, there are five such frameworks: Universal Composability (UC) by Canetti [Can00], the similar GNUC framework by Hofheinz and Shoup [HS11], Reactive Simulatability by Pfitzmann and Waidner [PW01], IITMs by Küsters [Küs06], and Abstract Cryptography by Maurer and Renner [MR11]. Even though the UC and GNUC frameworks differ in their mathematical formalism, they are essentially the same [HS11]. To understand this paper, it is sufficient to be familiar with either.

In the UC/GNUC framework, an abstract specification—often called the ideal functionality—describing the input and output behaviour of the protocol is given. A cryptographic protocol is then said to securely implement this ideal functionality, if an external adversary cannot distinguish between a run of the actual protocol and a run where the ideal functionality is performed by a trusted third party receiving the inputs and generating the ouputs for all parties. The protocol can now be used instead of the ideal functionality in any arbitrary complex system.

In this paper we make use of standard ideal functionalities: authenticated channels ($\mathcal{F}_{\text{ach}}$), secure channels ($\mathcal{F}_{\text{sch}}$), and zero-knowledge proofs ($\mathcal{F}_{\text{ZK}}$) as described in Section 12.1 of the GNUC paper [HS11]. The first two functionalities are essentially the same as Canetti's [Can00].

The $\mathcal{F}_{\mathrm{ZK}}$ functionality of GNUC differs from Canetti's definition in that the instance of the predicate to be proven is a private input of the prover, and is delivered to the verifier only in the last message of the protocol: this enables the prover to securely erase her witnesses before revealing the statement to be proven. We reproduce the formal definition of $\mathcal{F}_{\mathrm{ZK}}$ in Appendix A.1 for the reader's convenience.

We follow the formalism of GNUC to model common reference strings and system parameters—see Section 10 of the GNUC paper [HS11].

2.3 Zero-Knowledge Proofs of Existence

In the UC model, all proofs are necessarily proofs of *knowledge*. By embracing the extension to the UC model proposed by Camenisch, Krenn, and Shoup [CKS11], it becomes possible to perform proofs of *existence* in addition to proofs of *knowledge*. The former are computationally significantly less expensive. To that effect, the paper introduced the *gullible* zero-knowledge functionality $\mathcal{F}_{\mathrm{gZK}}$. Roughly speaking, $\mathcal{F}_{\mathrm{gZK}}$ is similar to the well-known zero-knowledge proof functionality $\mathcal{F}_{\mathrm{ZK}}$, except that not all the witnesses can be extracted. $\mathcal{F}_{\mathrm{gZK}}$ is not an ideal functionality in the UC/GNUC sense, but abstracts a concrete zero-knowledge proof protocol using secure channels $\mathcal{F}_{\mathrm{sch}}$ and a CRS.

When specifying the predicate to be proven, we will use the notation introduced by Camenisch, Krenn, and Shoup [CKS11] (which is very similar to the Camenisch-Stadler notation [CS97]); for example: $\aleph\alpha\ \exists\beta : \ y = g^{\beta} \wedge z = g^{\alpha}h^{\beta}$ is used for proving the existance of the discrete logarithm of y to the base g, and of a representation of z to the bases g and h such that the h-part of this representation is equal to the discrete logarithm of y to the base g. Furthermore, knowledge of the g-part of the representation (discrete logarithm of the Elgamal plaintext) is proven. Variables quantified by $\aleph$ can be extracted by the simulator in the security proof, while variables quantified by $\exists$ cannot.

In this paper, we will be proving statements involving encryptions and commitments, all of which can be easily translated into predicates of the form considered in Camenisch et al.'s paper [CKS11]. For predicates of this type, $\mathcal{F}_{\mathrm{gZK}}$ can be efficiently realized in the CRS model.

Ideal functionality $\mathcal{F}_{gZK}$. In Camenisch et al's paper, the $\mathcal{F}_{\mathrm{gZK}}$ ideal functionality was formally defined for the UC model, but one can easily port it to the GNUC model. We provide here only an informal description of $\mathcal{F}_{\mathrm{gZK}}$; see Appendix A.2 or the Camenisch et al. paper for the formal definition.

In the following we let R be a binary predicate that maps a triple (x, w_k, w_e) to 0 or 1, where x is called the *instance* and the pair (w_k, w_e) the *witness*. $\mathcal{F}_{\mathrm{gZK}}$ is parametrized by R and a leakage function ℓ (which for example reports the length of its input). The functionality also expects an arbitrary label to distinguish different proof instances.

The common input to $\mathcal{F}_{\mathrm{gZK}}$ is an arbitrary label. The prover's input is (x, w_k, w_e) where $R(x, w_k, w_e) = 1$. Next, $\mathcal{F}_{\mathrm{gZK}}$ leaks the length of the instance and witness $\ell(x, w_k)$ to the adversary $\mathcal{A}$. After an acknowledgement by $\mathcal{A}$, $\mathcal{F}_{\mathrm{gZK}}$ delivers the instance x to the verifier, while simultaneously erasing the witness (w_k, w_e). In the security proof, the simulator can extract w_k, but not w_e. Per convention, $\mathcal{F}_{\mathrm{gZK}}$ rejects malformed messages and messages with duplicate labels.

2.4 Homomorphic Semantically Secure Encryption

Definition. We define the key generation function $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(\mathsf{n})$, where n is a safe RSA modulus of unknown factorization. We define the encryption function $\mathsf{E} \leftarrow \mathsf{Enc}(v, \mathsf{pk}, \mathsf{r})$ that takes as input a plaintext v, a public key pk and some randomness r, and outputs a ciphertext E. We will also use the shorthand notation $(\mathsf{E}, \mathsf{r}) \xleftarrow{\$} \mathsf{Enc}(v, \mathsf{pk})$ in which the randomness r is chosen inside the Enc function. The corresponding decryption function $v' \leftarrow \mathsf{Dec}(\mathsf{E}, \mathsf{sk})$ takes as input the ciphertext and secret key, and outputs the plaintext. We assume that the encryption is homomorphic with respect to addition over $\mathbb{Z}_{\mathsf{n}}$: $\forall v_1, v_2 \in \mathbb{Z}_{\mathsf{n}}, \mathsf{r}_1, \mathsf{r}_2 : (\mathsf{pk}, \mathsf{sk}) \in \mathsf{KeyGen}(\mathsf{n}) \implies \mathsf{Dec}(\mathsf{Enc}(v_1, \mathsf{pk}, \mathsf{r}_1) * \mathsf{Enc}(v_2, \mathsf{pk}, \mathsf{r}_2), \mathsf{sk}) = v_1 + v_2$.

We require that correctness of encryption and decryption be efficiently provable with $\mathcal{F}_{\text{gZK}}$, and that it is possible to efficiently prove knowledge of sk given pk with $\mathcal{F}_{\text{gZK}}$. We will use a shorthand notation to denote such proofs, e.g.: $\nvDash \mathsf{sk}, v : (\mathsf{pk}, \mathsf{sk}) \in \mathsf{KeyGen}(\mathsf{n}) \wedge v = \mathsf{Dec}(\mathsf{E}, \mathsf{sk})$.

Camenisch-Shoup encryption. An example of such an encryption scheme is the simplified version of Camenisch-Shoup encryption [CS03,DJ03] with a short private key and short randomness, described by Jarecki and Shmatikov [JS07]. The key generation function is: $\mathsf{x} \xleftarrow{\$} \mathbb{Z}_{\lfloor \sqrt{\mathsf{n}} \rfloor}$, $\mathsf{g}' \xleftarrow{\$} \mathbb{Z}^*_{\mathsf{n}^2}$, $\mathsf{g} \leftarrow \mathsf{g}'^{2\mathsf{n}}$, $\mathsf{y} \leftarrow \mathsf{g}^{\mathsf{x}}$; the secret key sk is x, and the public key pk is (g, y). To encrypt the message $v \in \mathbb{Z}_{\mathsf{n}}$: $\mathsf{r} \xleftarrow{\$} \mathbb{Z}_{\lfloor \sqrt{\mathsf{n}} \rfloor}$, $\mathsf{u} \leftarrow \mathsf{g}^{\mathsf{r}}$, $\mathsf{e} \leftarrow \mathsf{y}^{\mathsf{r}}(\mathsf{n}+1)^v \pmod{\mathsf{n}^2}$; the ciphertext E is (u, e). To decrypt: $v''' \leftarrow (\mathsf{e}/\mathsf{u}^{\mathsf{x}})^2$, $v'' \leftarrow \frac{v'''-1}{\mathsf{n}}$ (over the integers), $v' \leftarrow v'' \cdot 2^{-1} \pmod{\mathsf{n}}$; output v'. This scheme is semantically secure if Paillier's Decision Composite Residuosity Assumption [Pai99] holds.

2.5 Mixed Trapdoor Commitment Scheme

We now construct a commitment scheme which we will use instead of traditional UC commitment schemes [CF01] in our circuit evaluation protocol. Our commitment scheme works well with proofs of *existence* using $\mathcal{F}_{\text{gZK}}$, resulting in an efficiency gain in the overall protocol.[4] To the best of our knowledge, this is a novel scheme.

We define a mixed trapdoor commitment scheme to be a commitment scheme that is either: perfectly hiding and equivocable; or statistically binding, depending on the distribution of the CRS. Mixed trapdoor commitments are similar to UC commitments [CF01] in that *1)* the simulator can equivocate commitments in the security proof without being caught, even if he has to provide all randomness used to generate the commitment to the adversary; and *2)* the simulator can use an adversary who equivocates commitments to solve a hard cryptographic problem. However unlike UC commitments, in mixed trapdoor commitments *3)* the simulator *does not need to extract the openings or the committed values* from $\mathcal{F}_{\text{gZK}}$.

Definition. Let $\mathfrak{cp}_i \xleftarrow{\$} \mathsf{ComGen}_i(\mathsf{n})$ for $i \in \{0, 1\}$ be functions that generate parameters for a commitment scheme. If $i = 0$, the commitment scheme is perfectly hiding (computationally binding), and if $i = 1$, the commitment scheme is statistically binding (computationally hiding). For the perfect-hiding setting, we define the function $(\mathfrak{cp}'_0, \mathfrak{t}) \xleftarrow{\$} \mathsf{ComGen}'_0(\mathsf{n})$ that additionally outputs a trapdoor $\mathfrak{t}$. We further require that $\mathfrak{cp}_0$, $\mathfrak{cp}'_0$, and $\mathfrak{cp}_1$ are pairwise computationally indistinguishable.

[4] The efficiency gain due to using proofs of existence instead of proofs of knowledge outweighs the efficiency loss due to the more complex commitment scheme.

We define the function $(\mathfrak{C}, \mathfrak{x}) \xleftarrow{\$} \mathsf{Com}_{\mathfrak{cp}_i}(v)$ that takes as input a value $v \in \mathbb{Z}_{\mathfrak{n}}$ to be committed, and outputs a commitment $\mathfrak{C}$ and an opening $\mathfrak{x}$ to the commitment. We will also use the notation $\mathfrak{C} \leftarrow \mathsf{Com}_{\mathfrak{cp}_i}(v, \mathfrak{x})$, where the opening is chosen outside of the function. Conversely, we define the verification function $\mathsf{ComVfy}_{\mathfrak{cp}_i}(\mathfrak{C}, \mathfrak{x}, v)$ that checks whether the tuple $(\mathfrak{C}, \mathfrak{x})$ is one of the possible values generated by $\mathsf{Com}_{\mathfrak{cp}_i}(v)$. The commitments are homomorphic with respect to addition over $\mathbb{Z}_{\mathfrak{n}}$: $\mathsf{ComVfy}_{\mathfrak{cp}_i}(\mathfrak{C}_1, \mathfrak{x}_1, v_1) \wedge \mathsf{ComVfy}_{\mathfrak{cp}_i}(\mathfrak{C}_2, \mathfrak{x}_2, v_2) \implies \mathsf{ComVfy}_{\mathfrak{cp}_i}(\mathfrak{C}_1 * \mathfrak{C}_2, \mathfrak{x}_1 + \mathfrak{x}_2, v_1 + v_2)$. With a trapdoor $\mathfrak{t}$ it is possible to efficiently equivocate commitments in the perfect-hiding setting: $\forall v' \in \mathbb{Z}_{\mathfrak{n}}$; $\mathfrak{x}' \leftarrow \mathsf{Trapdoor}_{\mathfrak{cp}'_0}(\mathfrak{t}, \mathfrak{C}, \mathfrak{x}, v, v')$: $\mathsf{ComVfy}_{\mathfrak{cp}'_0}(\mathfrak{C}, \mathfrak{x}, v) \implies \mathsf{ComVfy}_{\mathfrak{cp}'_0}(\mathfrak{C}, \mathfrak{x}', v')$.

We require that verifying a commitment be efficient with $\mathcal{F}_{\text{gZK}}$.

In the sequel, we drop the subscript $\mathfrak{cp}_i$ if it clear which parameters need to be used.

Construction based on Elgamal. We now provide the construction of a mixed trapdoor commitment scheme based on Elgamal encryption. We construct ComGen_1 as follows: *1)* find the first prime $\mathfrak{p}$ such that $\mathfrak{p} = k \cdot \mathfrak{n} + 1$ for some $k \in \mathbb{N}$— according to a heuristic[5] by Wagstaff [WJ79]: $\mathfrak{p} < \mathfrak{n} \cdot (\log \mathfrak{n})^2$; *2)* find a generator $\mathfrak{g}$ of a subgroup of $\mathbb{Z}_{\mathfrak{p}}$ of order $\mathfrak{n}$; *3)* select $\mathfrak{a}, \mathfrak{t}, \mathfrak{m}$ at random from $\mathbb{Z}_{\mathfrak{n}}$; *4)* compute $\mathfrak{h} \leftarrow \mathfrak{g}^{\mathfrak{a}}$, $\mathfrak{y} \leftarrow \mathfrak{g}^{\mathfrak{m}}\mathfrak{h}^{\mathfrak{t}}$, $\mathfrak{u} \leftarrow \mathfrak{g}^{\mathfrak{t}}$, i.e., $(\mathfrak{y}, \mathfrak{u})$ is the Elgamal encryption of $\mathfrak{g}^{\mathfrak{m}}$ with the public key $(\mathfrak{g}, \mathfrak{h})$; *5)* output $\mathfrak{cp}_1 \leftarrow (\mathfrak{p}, \mathfrak{g}, \mathfrak{h}, \mathfrak{y}, \mathfrak{u})$. In practice, where we want to select a random common reference string $\mathfrak{cp}_1$, it is also possible to randomly sample $\mathfrak{h}$, $\mathfrak{y}$, and $\mathfrak{u}$ from the subgroup generated by $\mathfrak{g}$. With high probability, we have that $\gcd(\mathfrak{a}, \mathfrak{n}) = \gcd(\mathfrak{m}, \mathfrak{n}) = \gcd(\mathfrak{t}, \mathfrak{n}) = 1$, which means that $\mathfrak{h}, \mathfrak{y}, \mathfrak{u}$ are all of order $\mathfrak{n}$. We construct ComGen_0 similarly, except that in step 3, we set $\mathfrak{m} \leftarrow 0$. The function ComGen'_0 additionally outputs $\mathfrak{t}$.

To commit to $v \in \mathbb{Z}_{\mathfrak{n}}$, one sets $\mathfrak{x} \xleftarrow{\$} \mathbb{Z}_{\mathfrak{n}}$; $\mathfrak{C}_1 \leftarrow \mathfrak{y}^v\mathfrak{h}^{\mathfrak{x}}$; $\mathfrak{C}_2 \leftarrow \mathfrak{u}^v\mathfrak{g}^{\mathfrak{x}}$; and $\mathfrak{C} \leftarrow (\mathfrak{C}_1, \mathfrak{C}_2)$. The latter is a re-randomized encryption of $\mathfrak{g}^{\mathfrak{m} \cdot v}$. Verification is trivial. Finally, if $\mathfrak{m} = 0$ and one knows the trapdoor information $\mathfrak{t}$, one can open the commitment $\mathfrak{C}$ to a different value $v' \in \mathbb{Z}_{\mathfrak{n}}$ by setting $\mathfrak{x}' \leftarrow (v - v') \cdot \mathfrak{t} + \mathfrak{x}$.

3 Our Ideal Functionality $\mathcal{F}_{\text{ABB}}$

In this section, we will start by giving a short informal definition of the ideal functionality $\mathcal{F}_{\text{ABB}}$ (arithmetic black box) for doing computation over $\mathbb{Z}_{\mathfrak{n}}$. We then provide the formal definition of $\mathcal{F}_{\text{ABB}}$ using the notation of GNUC [HS11]. It is not necessary to read Subsections 3.2 and 3.3 to understand the construction of our scheme.

3.1 Informal Definition of $\mathcal{F}_{\text{ABB}}$

The functionality $\mathcal{F}_{\text{ABB}}$ reacts to a set of instructions. Per convention, both parties must agree on the instruction and the shared input before $\mathcal{F}_{\text{ABB}}$ executes it. An instruction may require $\mathcal{P}$ and $\mathcal{Q}$ to send multiple messages to $\mathcal{F}_{\text{ABB}}$ in a specific order, however $\mathcal{F}_{\text{ABB}}$ may run other instructions concurrently while waiting for the next message. More precisely $\mathcal{P}$ and $\mathcal{Q}$ can: provide inputs to $\mathcal{F}_{\text{ABB}}$; ask it do to a linear combination or multiplication of previous inputs or intermediate results; ask it to output a value to one of them; and do an arbitrary zero-knowledge proof involving inputs/outputs to/from the circuit and external witnesses. These instructions can be arbitrarily interleaved, intermediate results output and new inputs be provided. The input values provided by $\mathcal{P}$ and $\mathcal{Q}$ may depend on output values obtained. Following the GNUC

[5] We confirmed this experimentally for 250 randomly generated 1248-bit safe RSA moduli.

formalism, each message sent to $\mathcal{F}_{\text{ABB}}$ is prefixed with a label which contains, among others, the name of the instruction to execute, the current step in the instruction this message refers to, and the shared input φ; the private inputs are always part of the message body.

State. The ideal functionality $\mathcal{F}_{\text{ABB}}$ is stateful. It maintains an associative array V, mapping identifiers (in $\Sigma^\star$) to integer values (in $\mathbb{Z}_{\mathsf{n}}$).

Instructions. These are the instructions supported by $\mathcal{F}_{\text{ABB}}$:

- *Input from $\mathcal{P}$:* $\mathcal{P}$'s private input is the value v. $\mathcal{F}_{\text{ABB}}$ parses the shared input φ as $\langle k \rangle$, where k will be the identifier associated to the value v, and sets $V[k] \leftarrow v$.
- *Input from $\mathcal{Q}$:* $\mathcal{Q}$'s private input is v. $\mathcal{F}_{\text{ABB}}$ parses φ as $\langle k \rangle$, and sets $V[k] \leftarrow v$.
- *Linear combination:* $\mathcal{F}_{\text{ABB}}$ parses φ as $\langle m, k_0, v_0, \langle k_1, v_1 \rangle, \ldots, \langle k_{m-1}, v_{m-1} \rangle \rangle$ and sets: $V[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} V[k_i] \cdot v_i$.
- *Multiplication:* $\mathcal{F}_{\text{ABB}}$ parses φ as $\langle k_0, k_1, k_2 \rangle$ and sets: $V[k_0] \leftarrow V[k_1] \cdot V[k_2]$.
- *Output to $\mathcal{P}$:* $\mathcal{F}_{\text{ABB}}$ parses φ as $\langle k \rangle$, and sends $V[k]$ (as a *delayed output*) to $\mathcal{P}$.
- *Output to $\mathcal{Q}$:* $\mathcal{F}_{\text{ABB}}$ parses φ as $\langle k \rangle$, and sends $V[k]$ (as a *delayed output*) to $\mathcal{Q}$.
- *Proof by $\mathcal{P}$:* This instruction can be used to prove a statement about values that were input/output to/from from the circuit ($\mathcal{F}_{\text{ABB}}$) and witnesses from a higher-level protocol. $\mathcal{P}$'s private input is $\langle x, w_k, \langle \rangle \rangle$. $\mathcal{F}_{\text{ABB}}$ parses φ as $\langle m, \langle k_0, \ldots, k_{m-1} \rangle, R \rangle$, where is R is a binary predicate that is compatible with $\mathcal{F}_{\text{gZK}}$ and which can involve *1)* values that were input by $\mathcal{P}$ to $\mathcal{F}_{\text{ABB}}$, *2)* values that were output to $\mathcal{P}$ from $\mathcal{F}_{\text{ABB}}$, and *3)* witnesses external to $\mathcal{F}_{\text{ABB}}$; x is an instance for R; w_k is a list of witnesses that are external to the circuit whose knowledge are proven; and $k_0, \ldots, k_{m-1}$ are identifiers of values in the circuit that were input by $\mathcal{P}$ or output to $\mathcal{P}$. $\mathcal{F}_{\text{ABB}}$ checks if the predicate holds, i.e., if $R\big(x,\ w_k \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$; and sends $\langle x \rangle$ (as a *delayed output*) to $\mathcal{Q}$. In Section 3.3, we define an extention of $\mathcal{F}_{\text{ABB}}$ denoted $\mathcal{F}_{\text{gABB}}$ which also allows for proofs of *existence* inside the *Proof* functionality.
- *Proof by $\mathcal{Q}$:* Similar to *Proof by $\mathcal{P}$* , with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.
- *Dynamic corruption:* $\mathcal{F}_{\text{ABB}}$ accepts a special corrupt message from $\mathcal{P}$ or $\mathcal{Q}$. From then on, all input and output of the corrupted party is redirected to the adversary $\mathcal{A}$, and $\mathcal{A}$ may recover all of the corrupted party's input (by asking $\mathcal{F}_{\text{ABB}}$ for it).

Treatment of invalid input. In case $\mathcal{F}_{\text{ABB}}$ receives a message it does not expect, a message that it cannot parse, or a message with a label it has seen previously from the same party, it simply ignores the message.

Comments. The value of n is not an input to $\mathcal{F}_{\text{ABB}}$, nor is it modeled as a CRS. Rather, it is modeled in the GNUC framework as a *system parameter*. Roughly speaking, this is a special type of ideal functionality to which all parties, including the environment, have common access. The value of n is generated by a trusted party, and no other party learns its factorization. Furthermore, the modulus n can be re-used across different protocol instances. In the setting of credential-authenticated identification [CCGS10] this is completely natural, as one can use a modulus generated by the credential issuer. In a different context, we can also imagine using the modulus n of a well-known and respected certificate authority (e.g., the modulus in Verisign's root certificate).

Our ideal functionality $\mathcal{F}_{\text{ABB}}$ shares some similarity with Nielsen's arithmetic black box (ABB) [Nie03], and Damgård and Orlandi's $\mathcal{F}_{\text{AMPC}}$ [DO10a]. The major difference is that our

$\mathcal{F}_{\text{ABB}}$ includes the *Proof* instruction, allowing values from higher-level protocols to be input and output securely. This instruction is crucial as it allows meaningful composition with other protocols (see Introduction). Unlike $\mathcal{F}_{\text{AMPC}}$, we do not support random number generation in the vanilla $\mathcal{F}_{\text{ABB}}$ for simplicity; see Section 5 of the full version [CES13] for an algorithm generating these that uses only our core set of instructions.

3.2 Formal Definition of $\mathcal{F}_{\text{ABB}}$

We now formally define the $\mathcal{F}_{\text{ABB}}$ functionality using the formalism of GNUC [HS11].

By $\langle \texttt{label}, \text{value1}, \text{value2}, \ldots \rangle$ we denote an ideal message with label `label` and payload value1, value2, ... By convention, if a party sends a message with the same label as a message it has sent previously, $\mathcal{F}_{\text{ABB}}$ ignores the message.

System parameters. The safe RSA modulus n, which defines the ring $\mathbb{Z}_{\mathsf{n}}$ in which all the arithmetic operations will be performed, and whose factorization is unknown to $\mathcal{P}$, $\mathcal{Q}$, and the adversary $\mathcal{A}$, is assumed to be part of the system parameters.

CRS. The CRS consists of the parameters cp of the commitment scheme.

State. The ideal functionality $\mathcal{F}_{\text{ABB}}$ is stateful, and maintains an associative array V, as well as the sets KPP, KQP, KPQ, KQQ, AP, AQ, RP, and RQ. The associative array V maps an identifier (in $\Sigma^\star$) to the corresponding value (in $\mathbb{Z}_{\mathsf{n}}$) in the circuit. The set KPP contains the list of identifiers corresponding to values that were either input by $\mathcal{P}$ or output to $\mathcal{P}$, in $\mathcal{P}$'s view; these identifiers can be used in the *Proof by* $\mathcal{P}$ instruction. The sets KQP, KPQ, KQQ are similar, but for $\mathcal{Q}$'s values in $\mathcal{P}$'s view, $\mathcal{P}$'s values in $\mathcal{Q}$'s view, $\mathcal{Q}$'s values in $\mathcal{Q}$'s view, respectively. The set AP contains the list of identifiers which, in $\mathcal{P}$'s view, have already been used in the circuit; this set prevents parties from using the same identifier multiple times. The set AQ is similar, but for $\mathcal{Q}$'s view. The set RP contains the list of identifiers which, in $\mathcal{P}$'s view, are ready to be used in other instructions; this set prevents parties from using an identifier where the corresponding value has not been properly initialized yet. The set RQ is similar, but for $\mathcal{Q}$'s view.

Instructions. In what follows, we let $\varphi \in \Sigma^\star$ denote the command ID, a string which will be part of the label. The command ID φ will contain all the common input to an instruction.

The ideal functionality $\mathcal{F}_{\text{ABB}}$ is composed of several instructions. By convention each step may be triggered only once (re-use of an instruction requires a different command ID φ). A logical expression in $[\ldots]$: is a guard that must be satisfied in order to trigger the step. Our ideal instructions are modelled closely after GNUC's zero-knowledge and secure function evaluation functionalities [HS11].

We take the convention that variables with an overbar, such as $\bar{v}$, are global variables associated with the command ID φ, whose scope is the instruction they are defined in. All other variables are local.

We also assume that the communication between the parties and $\mathcal{F}_{\text{ABB}}$ cannot be delayed by $\mathcal{A}$, this makes sure that in the case one party re-uses an output label k in several instructions, the operation that will be ignored by $\mathcal{F}_{\text{ABB}}$ is clearly defined. The ideal functionality $\mathcal{F}_{\text{ABB}}$ models message delays internally.

- **Input from $\mathcal{P}$:** In this instruction, parse the command ID φ as $\langle k \rangle$ where $k = \langle \varrho \rangle$ with $\varrho \in \Sigma^\star$.
  - $\texttt{ip:send:}\varphi$: Accept $\langle \texttt{ip:send:}\varphi, v \rangle$ from $\mathcal{P}$ where $v \in \mathbb{Z}_\mathsf{n}$ and the identifier is not yet assigned: $k \notin AP$. Mark the identifier as assigned: $AP \leftarrow k$. Set $\bar{v} \leftarrow v$. Send $\langle \texttt{ip:send:}\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{ip:ready:}\varphi$: Accept $\langle \texttt{ip:ready:}\varphi \rangle$ from $\mathcal{Q}$, where the identifier is not yet assigned: $k \notin AQ$. Mark the identifier as assigned: $AQ \leftarrow k$. Send $\langle \texttt{ip:ready:}\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{ip:lock:}\varphi$ $[\texttt{ip:send:}\varphi \wedge \texttt{ip:ready:}\varphi]$: Accept $\langle \texttt{ip:lock:}\varphi \rangle$ from $\mathcal{A}$. Store the identifier-value pair in the map V: $V[k] \leftarrow \bar{v}$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{ip:done:}\varphi$ $[\texttt{ip:lock:}\varphi]$: Accept $\langle \texttt{ip:done:}\varphi \rangle$ from $\mathcal{A}$. Mark the value as being ready to be used by other instructions: $RP \leftarrow k$. Mark the value as being known to $\mathcal{P}$: $KPP \leftarrow k$. Send $\langle \texttt{ip:done:}\varphi \rangle$ to $\mathcal{P}$.
  - $\texttt{ip:deliver:}\varphi$ $[\texttt{ip:lock:}\varphi]$: Accept $\langle \texttt{ip:deliver:}\varphi \rangle$ from $\mathcal{A}$. Mark the value as being ready to be used by other instructions: $RQ \leftarrow k$. Mark the value as being known to $\mathcal{P}$: $KPQ \leftarrow k$. Send $\langle \texttt{ip:deliver:}\varphi \rangle$ to $\mathcal{Q}$.
  - $\texttt{ip:reset:}\varphi$ $[\neg\texttt{ip:lock:}\varphi \wedge \texttt{corrupt:P}]$: Accept $\langle \texttt{ip:reset:}\varphi, v \rangle$ from $\mathcal{A}$. Replace $\mathcal{P}$'s input by $\mathcal{A}$'s: $\bar{v} \leftarrow v$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{ip:expose:}\varphi$ $[\texttt{ip:send:}\varphi \wedge \texttt{corrupt:P}]$: Accept $\langle \texttt{ip:expose:}\varphi \rangle$ from $\mathcal{A}$. Send $\langle \texttt{ip:expose:}\varphi, \bar{v} \rangle$ to $\mathcal{A}$.
- **Input from $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `iq`. We do not formalize this instruction here.
- **Output to $\mathcal{P}$:** In this instruction, parse φ as $\langle k \rangle$ where $k \in \Sigma^\star$.
  - $\texttt{op:p:}\varphi$: Accept message $\langle \texttt{op:p:}\varphi \rangle$ from $\mathcal{P}$, where the value is ready to be used: $k \in RP$. Send $\langle \texttt{op:p:}\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{op:q:}\varphi$: Accept message $\langle \texttt{op:q:}\varphi \rangle$ from $\mathcal{Q}$, where the value is ready to be used: $k \in RQ$. Send $\langle \texttt{op:q:}\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{op:lock:}\varphi$ $[\texttt{op:p:}\varphi \wedge \texttt{op:q:}\varphi]$: Accept $\langle \texttt{op:lock:}\varphi \rangle$ from $\mathcal{A}$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{op:deliver:}\varphi$ $[\texttt{op:lock:}\varphi]$: Accept $\langle \texttt{op:deliver:}\varphi \rangle$ from $\mathcal{A}$. Mark the value as being known to $\mathcal{P}$: $KPP \leftarrow k$. Send $\langle \texttt{op:deliver:}\varphi, V[k] \rangle$ to $\mathcal{P}$.
  - $\texttt{op:done:}\varphi$ $[\texttt{op:lock:}\varphi]$: Accept $\langle \texttt{op:done:}\varphi \rangle$ from $\mathcal{A}$. Mark the value as being known to $\mathcal{P}$: $KPQ \leftarrow k$. Send $\langle \texttt{op:done:}\varphi \rangle$ to $\mathcal{Q}$.
- **Output to $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `oq`. We do not formalize this instruction here.
- **Linear combination:** In this instruction, parse φ as $\langle m, k_0, v_0, \langle k_1, v_1 \rangle, \ldots, \langle k_{m-1}, v_{m-1} \rangle \rangle$ where $m \in \mathbb{N}^*$, $\forall i \in \mathbb{N}_m : k_i \in \Sigma^\star, v_i \in \mathbb{Z}_\mathsf{n}$, and where $k_0 = \langle \texttt{joint}, \varrho \rangle$ with $\varrho \in \Sigma^\star$.
  - $\texttt{l:p:}\varphi$: Accept message $\langle \texttt{l:p:}\varphi \rangle$ from $\mathcal{P}$ where the zeroth (output) identifier is not yet assigned: $k_0 \notin AP$, and the other identifiers are ready to be used: $\forall i \in \mathbb{N}_m^* : k_i \in RP$. Mark the output identifier as being assigned: $AP \leftarrow k_0$. Send $\langle \texttt{l:p:}\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{l:q:}\varphi$: Accept message $\langle \texttt{l:q:}\varphi \rangle$ from $\mathcal{Q}$ where the zeroth (output) identifier is not yet assigned: $k_0 \notin AQ$, and the other identifiers are ready to be used: $\forall i \in \mathbb{N}_m^* : k_i \in RQ$. Mark the output identifier as being assigned: $AQ \leftarrow k_0$. Send $\langle \texttt{l:q:}\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{l:lock:}\varphi$ $[\texttt{l:p:}\varphi \wedge \texttt{l:q:}\varphi]$: Accept message $\langle \texttt{l:lock:}\varphi \rangle$ from $\mathcal{A}$. Store the result in the map V: $V[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} V[k_i] \cdot v_i$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{l:done:p:}\varphi$ $[\texttt{l:lock:}\varphi]$: Accept $\langle \texttt{l:done:p:}\varphi \rangle$ from $\mathcal{A}$. Mark the output identifier as being ready to be used by other instructions: $RP \leftarrow k_0$. Send $\langle \texttt{l:done:p:}\varphi \rangle$ to $\mathcal{P}$.

– $\texttt{l:done:q}:\varphi$ [$\texttt{l:lock}:\varphi$] : Accept $\langle \texttt{l:done:q}:\varphi \rangle$ from $\mathcal{A}$. Mark the output identifier as being ready to be used by other instructions: $RQ \leftarrow k_0$. Send $\langle \texttt{l:done:q}:\varphi \rangle$ to $\mathcal{Q}$.

- **Multiplication:** In this instruction, parse φ as $\langle k_0, k_1, k_2 \rangle$ where $k_1, k_2 \in \Sigma^\star$ and $k_0 = \langle \texttt{joint}, \varrho \rangle$ with $\varrho \in \Sigma^\star$.
  - $\texttt{m:p}:\varphi$: Accept message $\langle \texttt{m:p}:\varphi \rangle$ from $\mathcal{P}$ where the zeroth (output) identifier is not yet assigned: $k_0 \notin AP$, and the other identifiers are ready to be used: $\forall i \in \mathbb{N}_3^* : k_i \in RP$. Mark the output identifier as being assigned: $AP \leftarrow k_0$. Send $\langle \texttt{m:p}:\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{m:q}:\varphi$: Accept message $\langle \texttt{m:q}:\varphi \rangle$ from $\mathcal{Q}$ where the zeroth (output) identifier is not yet assigned: $k_0 \notin AQ$, and the other identifiers are ready to be used: $\forall i \in \mathbb{N}_3^* : k_i \in RQ$. Mark the output identifier as being assigned: $AQ \leftarrow k_0$. Send $\langle \texttt{m:q}:\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{m:lock}:\varphi$ [$\texttt{m:p}:\varphi \wedge \texttt{m:q}:\varphi$] : Accept message $\langle \texttt{m:lock}:\varphi \rangle$ from $\mathcal{A}$. Store the result in the map V: $V[k_0] \leftarrow V[k_1] \cdot V[k_2]$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{m:done:p}:\varphi$ [$\texttt{m:lock}:\varphi$] : Accept $\langle \texttt{m:done:p}:\varphi \rangle$ from $\mathcal{A}$. Mark the output identifier as being ready to be used by other instructions: $RP \leftarrow k_0$. Send $\langle \texttt{m:done:p}:\varphi \rangle$ to $\mathcal{P}$.
  - $\texttt{m:done:q}:\varphi$ [$\texttt{m:lock}:\varphi$] : Accept $\langle \texttt{m:done:q}:\varphi \rangle$ from $\mathcal{A}$. Mark the output identifier as being ready to be used by other instructions: $RQ \leftarrow k_0$. Send $\langle \texttt{m:done:q}:\varphi \rangle$ to $\mathcal{Q}$.
- **Proof by $\mathcal{P}$:** In this instruction, parse φ as $\langle m, \langle k_0, \ldots, k_{m-1} \rangle, R \rangle$, where $m \in \mathbb{N}$, $\forall i \in \mathbb{N}_m : k_i \in \Sigma^\star$, and where is R is a binary predicate that is compatible with $\mathcal{F}_{\text{gZK}}$.
  - $\texttt{pp:send}:\varphi$: Accept $\langle \texttt{pp:send}:\varphi, x, \tilde{m}, w_k, 0, \langle\rangle \rangle$ from $\mathcal{P}$ where x is an instance for R, $\tilde{m} \in \mathbb{N}$, $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1} \rangle$ is a list of external witnesses whose knowledge is proven, $R\big(x,\ (w_{k,0}, \ldots, w_{k,\tilde{m}-1}) \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$, and where all values in the common list are known to $\mathcal{P}$: $\forall i \in \mathbb{N}_m : k_i \in KPP$. Store the instance and all witnesses: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. Send $\langle \texttt{pp:send}:\varphi, \ell(x, w_k) \rangle$ to $\mathcal{A}$.
  - $\texttt{pp:ready}:\varphi$: Accept $\langle \texttt{pp:ready}:\varphi \rangle$ from $\mathcal{Q}$, where all values in the common list are known to $\mathcal{P}$: $\forall i \in \mathbb{N}_m : k_i \in KPQ$. Send $\langle \texttt{pp:ready}:\varphi \rangle$ to $\mathcal{A}$.
  - $\texttt{pp:lock}:\varphi$ [$\texttt{pp:send}:\varphi \wedge \texttt{pp:ready}:\varphi$] : Accept $\langle \texttt{pp:lock}:\varphi \rangle$ from $\mathcal{A}$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{pp:done}:\varphi$ [$\texttt{pp:lock}:\varphi$] : Accept $\langle \texttt{pp:done}:\varphi \rangle$ from $\mathcal{A}$. Send $\langle \texttt{pp:done}:\varphi \rangle$ to $\mathcal{P}$.
  - $\texttt{pp:deliver}:\varphi$ [$\texttt{pp:lock}:\varphi$] : Accept $\langle \texttt{pp:deliver}:\varphi, L \rangle$ from $\mathcal{A}$, where $L = \ell(\bar{x}, \bar{w}_k) \vee$ [$\texttt{corrupt:Q}$]. Send $\langle \texttt{pp:deliver}:\varphi, \bar{x} \rangle$ to $\mathcal{Q}$.
  - $\texttt{pp:reset}:\varphi$ [$\neg\texttt{pp:lock}:\varphi \wedge \texttt{corrupt:P}$] : Accept $\langle \texttt{pp:reset}:\varphi, x, \tilde{m}, w_k, 0, \langle\rangle \rangle$ from $\mathcal{A}$ where x is an instance for R, $\tilde{m} \in \mathbb{N}$, $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1} \rangle$ is a list of witnesses, and where $R\big(x,\ (w_{k,0}, \ldots, w_{k,\tilde{m}-1}) \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$. Store the instance and all witnesses: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. Send $\langle\rangle$ to $\mathcal{A}$.
  - $\texttt{pp:expose}:\varphi$ [$\texttt{pp:send}:\varphi \wedge \neg\texttt{pp:lock}:\varphi \wedge \texttt{corrupt:P}$] : Accept $\langle \texttt{pp:expose}:\varphi \rangle$ from $\mathcal{A}$. Send $\langle \texttt{pp:expose}:\varphi, \bar{x}, \bar{w}_k \rangle$ to $\mathcal{A}$.
- **Proof by $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to $\texttt{pq}$. We do not formalize this instruction here.
- **Dynamic corruption:**
  - $\texttt{corrupt:P}$: Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{P}$. Send $\langle \texttt{corrupt:P} \rangle$ to $\mathcal{A}$ together with an invitation for the messages $\langle \texttt{ip:expose}:\varphi \rangle$ (for all φ where the $\texttt{ip:send}:\varphi$ step has been processed already) and $\langle \texttt{pp:expose}:\varphi \rangle$ (for all φ where the $\texttt{pp:send}:\varphi$ step has been processed already).
  - $\texttt{corrupt:Q}$: Analogously, but for $\mathcal{Q}$.

Invalid input. In case $\mathcal{F}_{\text{ABB}}$ receives a message it does not expect, a message that it cannot parse, or a message with a label it has seen previously from the same party, it simply ignores the message.

3.3 Allowing proof of existence of external variables: $\mathcal{F}_{\text{gABB}}$

To improve the efficiency of higher-level protocols, protocol designers may wish to perform proofs of *existence* of external witnesses in the *Proof* instruction. This efficiency gain comes at the cost of additional complexity, this is why we decided to define a separate ideal functionnality called $\mathcal{F}_{\text{gABB}}$ that allows that feature.

Similarily to $\mathcal{F}_{\text{gZK}}$, $\mathcal{F}_{\text{gABB}}$ is not a proper ideal functionality in the UC sense, but rather a *gullible* ideal functionality as described by Camenisch et al. [CKS11]. This means that $\mathcal{F}_{\text{gABB}}$ does not check the correctness of the predicate inside the *Proof* instruction. The intended functionality of $\mathcal{F}_{\text{gABB}}$ is thus only guaranteed for so-called *nice* environments, i.e., environments which never ask $\mathcal{F}_{\text{gABB}}$ to prove a false statement. Roughly speaking, the special composition theorm of Camenisch et al. guarantees that when in a higher-level protocol $\mathcal{F}_{\text{gABB}}$ is replaced by the intended realization Π_{gABB}, the higher-level protocol is secure against *all* environments.

Informal definition of $\mathcal{F}_{gABB}$. The high-level description of the *Proof* instructions of $\mathcal{F}_{\text{gABB}}$ is the following:

- *Proof by $\mathcal{P}$:* $\mathcal{P}$'s private input is now $\langle x, w_k, w_e\rangle$, where w_e is a lists of witnesses that are external to the circuit whose existence is proven. The predicate is said to hold if $R\big(x,\ w_k \cup w_e \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$. However, $\mathcal{F}_{\text{gABB}}$, being a *gullible* functionality, does not check if the predicate holds.
- *Proof by $\mathcal{Q}$:* Similar to *Proof by $\mathcal{P}$* , with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.
  All other instructions are identical to the ones of $\mathcal{F}_{\text{ABB}}$.

Formal definition of $\mathcal{F}_{gABB}$. The formal definition of the *Proof* instructions of $\mathcal{F}_{\text{gABB}}$ is the following:

- **Proof by $\mathcal{P}$:** In this instruction, parse φ as $\langle m, \langle k_0, \ldots, k_{m-1}\rangle, R\rangle$, where $m \in \mathbb{N}$, $\forall i \in \mathbb{N}_m : k_i \in \Sigma^\star$, and where is R is a binary predicate that is compatible with $\mathcal{F}_{\text{gZK}}$.
  - `pp:send:`φ : Accept $\langle$`pp:send:`$\varphi, x, \tilde{m}, w_k, \hat{m}, w_e\rangle$ from $\mathcal{P}$ where x is an instance for R, $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1}\rangle$ is the list of external witnesses whose knowledge is proven, $w_e = \langle w_{e,0}, \ldots, w_{e,\hat{m}-1}\rangle$ is the list of external witnesses whose existence is proven, and where all values in the common list are known to $\mathcal{P}$: $\forall i \in \mathbb{N}_m : k_i \in KPP$. The ideal functionality $\mathcal{F}_{\text{gABB}}$, being *gullible*, does not check if the predicate holds. Store the instance and all witnesses quantified by $\forall$: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. Send $\langle$`pp:send:`$\varphi, \ell(x, w_k)\rangle$ to $\mathcal{A}$.
  - `pp:ready:`φ : Accept $\langle$`pp:ready:`$\varphi\rangle$ from $\mathcal{Q}$, where all values in the common list are known to $\mathcal{P}$: $\forall i \in \mathbb{N}_m : k_i \in KPQ$. Send $\langle$`pp:ready:`$\varphi\rangle$ to $\mathcal{A}$.
  - `pp:lock:`φ [`pp:send:`$\varphi \wedge$ `pp:ready:`φ] : Accept $\langle$`pp:lock:`$\varphi\rangle$ from $\mathcal{A}$. Send $\langle\rangle$ to $\mathcal{A}$.
  - `pp:done:`φ [`pp:lock:`φ] : Accept $\langle$`pp:done:`$\varphi\rangle$ from $\mathcal{A}$. Send $\langle$`pp:done:`$\varphi\rangle$ to $\mathcal{P}$.
  - `pp:deliver:`φ [`pp:lock:`φ] : Accept $\langle$`pp:deliver:`$\varphi, L\rangle$ from $\mathcal{A}$, where $L = \ell(\bar{x}, \bar{w}_k) \vee$ [`corrupt:Q`]. Send $\langle$`pp:deliver:`$\varphi, \bar{x}\rangle$ to $\mathcal{Q}$.
  - `pp:reset:`φ [$\neg$`pp:lock:`$\varphi \wedge$`corrupt:P`] : Accept $\langle$`pp:reset:`$\varphi, x, \tilde{m}, w_k, \hat{m}, w_e\rangle$ from $\mathcal{A}$, where x is an instance for R, $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1}\rangle$, and $w_e = \langle w_{e,0}, \ldots, w_{e,\hat{m}-1}\rangle$. The ideal

functionality $\mathcal{F}_{\text{gABB}}$, being *gullible*, does not check if the predicate holds. Store the instance and all witnesses quantified by $\aleph$: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. Send $\langle\rangle$ to $\mathcal{A}$.

– $\texttt{pp:expose:}\varphi$ [$\texttt{pp:send:}\varphi \wedge \neg\texttt{pp:lock:}\varphi \wedge \texttt{corrupt:P}$] : Accept $\langle\texttt{pp:expose:}\varphi\rangle$ from $\mathcal{A}$. Send $\langle\texttt{pp:expose:}\varphi, \bar{x}, \bar{w}_k\rangle$ to $\mathcal{A}$.

- **Proof by $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `pq`. We do not formalize this instruction here.

4 Construction

We now show how to construct a protocol Π_{ABB} for circuit evaluation modulo n. Our protocol uses two ideal functionalities: $\mathcal{F}_{\text{ach}}$ (authenticated channels) and $\mathcal{F}_{\text{gZK}}$ (zero-knowledge proofs). Additionally, we make use of a system parameter, the modulus n of unknown factorization; and a CRS, consisting of the output of ComGen_1 (statistically-binding commitment).

High-level idea. The high-level idea of our construction is that $\mathcal{P}$ and $\mathcal{Q}$ generate additive shares of all the values (inputs and intermediate results) in the circuit. Identifiers are used to keep track of the values and the cryptographic objects associated with them. Like for $\mathcal{F}_{\text{ABB}}$, parties agree on the instruction to be performed by sending a message containing an identical instruction name and identical common input to the protocol Π_{ABB}. The instructions of Π_{ABB} are implemented as follows: *Input* is achieved by one party setting her share to the input, and generating a commitment to that share; the other party sets his share to zero. *Output* is achieved by one party sending her share to the other party. For the *Linear combination* instruction, each party does a linear combination of their shares locally. For the *Multiplication* instruction, we make use of two instances of a 2-party subroutine Π_{mul}: on $\mathcal{P}$'s input a, and $\mathcal{Q}$'s input b, Π_{mul} outputs u to $\mathcal{P}$ and v to $\mathcal{Q}$ such that $u + v = a \cdot b$. The *Proof* instruction can be done with the help of a zero-knowledge proof functionality $\mathcal{F}_{\text{gZK}}$. To ensure security against malicious adversaries, both parties update the commitments to the shares in each instruction, and prove in zero-knowledge that all their computations were done honestly.

The Π_{mul} subroutine makes use of a homomorphic (modulo n), semantically secure, public-key encryption scheme, along with our mixed trapdoor commitment scheme. To achieve security against adaptive corruptions, new encryption/decryption keys need to be generated for every multiplication. To do this in a practical way, we use the semantically secure version of Camenisch-Shoup encryption [CS03,DJ03,JS07] with a short private key and short randomness, as described in Section 2.4. One key feature of this scheme is that key generation is fast: just a single exponentiation modulo n^2. Another key feature is that many encryption/decryption keys can be used in conjunction with the same n, which is crucial. Our commitment scheme is also used extensively in the overall protcol. We use the construction presented in Section 2.5 and work in the group of integers modulo a prime of the form $k \cdot \mathsf{n} + 1$. The homomorphic properties of the commitment scheme makes this choice of prime particularly useful and practical. Another tool we make heavy use of is UC zero-knowledge. Because of the proposed implementations of encryption and commitment schemes, these proof systems can all be implemented using the approach proposed by Camenisch et al. [CKS11]. Because the encryption and commitment schemes are both homomorphic modulo n, all of our cryptographic tools work very well together, and yield quite practical protocols. We also stress that our protocols are designed in a modular way: they only make use of these abstract primitives, and not of *ad hoc* algebraic constructions.

$\mathcal{P}$ **proceeds as follows**: $\mathcal{P}$'s input is $\langle \varphi, v \rangle$ with $v \in \mathbb{Z}_{\mathsf{n}}$.	$\mathcal{Q}$ **proceeds as follows**: $\mathcal{Q}$'s input is $\langle \varphi \rangle$.
Parse φ as $\langle k \rangle$ with $k \in \Sigma^\star$. Abort if $k \in AP$. Mark the identifier as assigned: $AP \leftarrow k$. Set shares: $SP[k] \leftarrow v$ and $SQ[k] \leftarrow 0$. Commit to share: $(CP[k], XP[k]) \xleftarrow{\$} \mathsf{Com}(v)$.	Parse φ as $\langle k \rangle$ with $k \in \Sigma^\star$. Abort if $k \in AQ$. Mark the identifier as assigned: $AQ \leftarrow k$. Set own share: $SQ[k] \leftarrow 0$. Commit: $CQ[k] \leftarrow \mathsf{Com}(0,0); XQ[k] \leftarrow 0$.
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\text{gZK}}$ with label $\langle \mathtt{ip}, \varphi \rangle$: $\mathsf{K}v \; \exists XP[k] : \mathsf{ComVfy}(CP[k], XP[k], v)$. The value $CP[k]$ is delivered to $\mathcal{Q}$ via $\mathcal{F}_{\text{gZK}}$.	
Set other's commitment: $CQ[k] \leftarrow \mathsf{Com}(0,0)$. Mark value as ready: $RP \leftarrow k$. Mark it as known: $KP \leftarrow k$.	Mark value as ready: $RQ \leftarrow k$. Mark it as known by $\mathcal{P}$: $KP \leftarrow k$.

Fig. 1: Input from $\mathcal{P}$.

4.1 Realizing Π_{ABB}

$\mathcal{P}$ and $\mathcal{Q}$ each maintain the following global state: several associative arrays mapping the identifier of a value in the circuit (in $\Sigma^\star$) to a variety of cryptographic objects: SP and SQ map to the shares of $\mathcal{P}$ and $\mathcal{Q}$ of the values in the circuit (in $\mathbb{Z}_{\mathsf{n}}$), respectively; CP and CQ map to the commitment of the corresponding shares; XP (maintained by $\mathcal{P}$ only) and XQ ($\mathcal{Q}$ only) map to the opening of the commitments. For the *Proof* functionality, both parties maintain lists of identifiers corresponding to values that are known to $\mathcal{P}$ and $\mathcal{Q}$: KP and KQ, respectively. Additionally, to ensure "thread-safety", they also maintain: lists of assigned identifiers AP ($\mathcal{P}$ only) and AQ ($\mathcal{Q}$ only) to avoid assigning the same identifier to several variables; and lists of identifiers RP ($\mathcal{P}$ only) and RQ ($\mathcal{Q}$ only) corresponding to values that are ready to be used in other instructions. The array that one would obtain by summing the entries of SP and SQ corresponding to values that are ready (i.e., $\{(k, v) | k \in RP \cap RQ \wedge v = SP[k] + SQ[k]\}$), corresponds to the array V of the ideal functionality, that maps identifiers to values in the circuit.

All other variables that we will introduce are local to one instance of a instruction or an instance of the Π_{mul} subroutine. Several instructions may be active at the same time, however we assume (following the GNUC model) that all operations performed during an activation (the time interval between starting to process a new input message and sending a message to another functionality) happen atomically.

Input from $\mathcal{P}$. In this instruction, $\mathcal{P}$ inputs a value v into the circuit and associates it with the identifier k: $\mathcal{P}$ sets her own share to v, and $\mathcal{Q}$ sets his share to 0. Then $\mathcal{P}$ generates a commitment to her share, which she sends (along with proof) to $\mathcal{Q}$. See Figure 1 for the construction.

Input from $\mathcal{Q}$. Similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

Output to $\mathcal{Q}$. In this instruction, $\mathcal{Q}$ retrieves the value identified by k from the circuit: $\mathcal{P}$ sends her share to $\mathcal{Q}$ together with a proof of correctness. See Figure 2.

Output to $\mathcal{P}$. Similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

Linear combination. In this instruction, a linear combination of values in the circuit (plus an optional constant) is computed: $V[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} V[k_i] \cdot v_i$. Concretely, both parties perform local operations on their shares. Additionally, $\mathcal{P}$ sends an empty message to $\mathcal{Q}$ to ensure that both parties agree on the shared input φ. See Figure 3.

$\mathcal{P}$ **proceeds as follows**:	$\mathcal{Q}$ **proceeds as follows**:
Both parties' input is $\langle\varphi\rangle$. It is parsed as $\varphi = \langle k\rangle$ with $k \in \Sigma^\star$.	
Wait until $k \in RP$.	Wait until $k \in RQ$.
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\text{gZK}}$ with label $\langle \mathtt{oq}, \varphi\rangle$: $\exists XP[k] : \mathsf{ComVfy}(CP[k], XP[k], SP[k])$. The value $SP[k]$ is delivered to $\mathcal{Q}$ via $\mathcal{F}_{\text{gZK}}$.	
Mark value as known to $\mathcal{Q}$: $KQ \leftarrow k$.	Save $SP[k]$, and mark as known: $KQ \leftarrow k$. $\mathcal{Q}$ **returns** $(SP[k] + SQ[k])$.

Fig. 2: Output to $\mathcal{Q}$.

$\mathcal{P}$ **proceeds as follows**:	$\mathcal{Q}$ **proceeds as follows**:
Both parties' input is $\langle\varphi\rangle$. It is parsed as $\varphi = \langle m, k_0, v_0, \langle k_1, v_1\rangle, \ldots, \langle k_{m-1}, v_{m-1}\rangle\rangle$ with $m \in \mathbb{N}^*$, $\forall i \in \mathbb{N}_m : k_i \in \Sigma^\star$ and $\forall i \in \mathbb{N}_m : v_i \in \mathbb{Z}_{\mathsf{n}}$.	
Abort if $k_0 \in AP$. Mark identifier: $AP \leftarrow k_0$. Wait until $\forall i \in \mathbb{N}_m : k_i \in RP$. Update own share: $SP[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} SP[k_i] \cdot v_i$; com.: $CP[k_0] \leftarrow \mathsf{Com}(v_0, 0) * \prod_{i=1}^{m-1} CP[k_i]^{v_i}$; opening: $XP[k_0] \leftarrow \sum_{i=1}^{m-1} XP[k_i] \cdot v_i$; $\mathcal{Q}$'s commitment: $CQ[k_0] \leftarrow \prod_{i=1}^{m-1} CQ[k_i]^{v_i}$.	Abort if $k_0 \in AQ$. Mark identifier: $AQ \leftarrow k_0$. Wait until $\forall i \in \mathbb{N}_m : k_i \in RQ$. Update own share: $SQ[k_0] \leftarrow \sum_{i=1}^{m-1} SQ[k_i] \cdot v_i$; commitment: $CQ[k_0] \leftarrow \prod_{i=1}^{m-1} CQ[k_i]^{v_i}$; opening: $XQ[k_0] \leftarrow \sum_{i=1}^{m-1} XQ[k_i] \cdot v_i$; $\mathcal{P}$'s c.: $CP[k_0] \leftarrow \mathsf{Com}(v_0, 0) * \prod_{i=1}^{m-1} CP[k_i]^{v_i}$.
$\mathcal{P}$ sends the empty string to $\mathcal{Q}$ using $\mathcal{F}_{\text{ach}}$ with label $\langle \mathtt{1}, \varphi\rangle$ to ensure that they agree on φ.	
Mark value as ready: $RP \leftarrow k_0$.	Mark value as ready: $RQ \leftarrow k_0$.

Fig. 3: Linear combination.

Multiplication. In this instruction, the product of two values in the circuit is computed: $V[k_0] \leftarrow V[k_1] \cdot V[k_2]$. We can rewrite this as:

$$SP[k_0] + SQ[k_0] \leftarrow \underbrace{SP[k_1]\cdot SP[k_2]}_{\hat{p}} + \underbrace{SP[k_1]\cdot SQ[k_2]}_{(\tilde{u}+\tilde{v})} + \underbrace{SQ[k_1]\cdot SP[k_2]}_{(u+v)} + \underbrace{SQ[k_1]\cdot SQ[k_2]}_{\hat{q}}$$

where we introduce $\hat{p}, \hat{q}, \tilde{u}, \tilde{v}, u, v$ to simplify the discussion. The idea of this protocol is for $\mathcal{P}$ and $\mathcal{Q}$ to compute $\hat{p}$ and $\hat{q}$, respectively, using their private shares. They then jointly compute $\tilde{u}$ and $\tilde{v}$ using the Π_{mul} subroutine, which we introduce for clarity and which we describe in Section 4.3. Afterwards, u and v are computed using a second instantiation of Π_{mul}. Finally, $\mathcal{P}$ sets $SP[k_0] \leftarrow \hat{p} + \tilde{u} + u$ and $\mathcal{Q}$ sets $SQ[k_0] \leftarrow \hat{q} + \tilde{v} + v$. See Figure 4 for the construction.

One can optimize the protocol in Figure 4 by using the same homomorphic encryption key for both instances of Π_{mul} and merging the proofs inside and outside of Π_{mul} whenever possible.[6] We can thus save one proof of correctess for the encryption key, and save on some overhead in $\mathcal{F}_{\text{gZK}}$.

Proof by $\mathcal{P}$. In this instruction, $\mathcal{P}$ proves to $\mathcal{Q}$ in zero-knowledge some statement involving *1)* witnesses outside of the circuit, *2)* values that $\mathcal{P}$ input into the circuit, and *3)* values that $\mathcal{P}$ got as an output from the circuit. Since we cannot do proofs of *existence* of external variables in Π_{ABB}, we require that the list w_e of witnesses whose *existence* is proven is empty. This list will be used only in Π_{gABB}. See Figure 5 for the construction.

Proof by $\mathcal{Q}$. Similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

[6] Concretely, one would merge the proofs with the following labels: *1)* $\langle \mathtt{m5}, \varphi\rangle$, $\langle \mathtt{cm1}, \langle \mathtt{m7}, \varphi\rangle\rangle$ and $\langle \mathtt{cm1}, \langle \mathtt{m8}, \varphi\rangle\rangle$; *2)* $\langle \mathtt{m6}, \varphi\rangle$, $\langle \mathtt{cm2}, \langle \mathtt{m7}, \varphi\rangle\rangle$, and $\langle \mathtt{cm2}, \langle \mathtt{m8}, \varphi\rangle\rangle$; *3)* $\langle \mathtt{cm3}, \langle \mathtt{m7}, \varphi\rangle\rangle$ and $\langle \mathtt{cm3}, \langle \mathtt{m8}, \varphi\rangle\rangle$; *4)* $\langle \mathtt{cm4}, \langle \mathtt{m7}, \varphi\rangle\rangle$ and $\langle \mathtt{cm4}, \langle \mathtt{m8}, \varphi\rangle\rangle$.

$\mathcal{P}$ **proceeds as follows**:	$\mathcal{Q}$ **proceeds as follows**:
Both parties' input is $\langle\varphi\rangle$. It is parsed as $\varphi = \langle k_0, k_1, k_2\rangle$ with $k_0, k_1, k_2 \in \Sigma^\star$.	
Abort if $k_0 \in AP$. Mark identifier as assigned: $AP \leftarrow k_0$. Wait until $k_1, k_2 \in RP$. $\hat{p} \leftarrow SP[k_1] \cdot SP[k_2]$; $(\mathfrak{C}_{\hat{p}}, \mathfrak{x}_{\hat{p}}) \xleftarrow{\$} \mathsf{Com}(\hat{p})$.	Abort if $k_0 \in AQ$. Mark identifier as assigned: $AQ \leftarrow k_0$. Wait until $k_1, k_2 \in RQ$. $\hat{q} \leftarrow SQ[k_1] \cdot SQ[k_2]$; $(\mathfrak{C}_{\hat{q}}, \mathfrak{x}_{\hat{q}}) \xleftarrow{\$} \mathsf{Com}(\hat{q})$.
The instructions in the next four rows can be run in parallel in multiple threads.	
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\text{gZK}}$ with label $\langle\mathtt{m5},\varphi\rangle$: $\exists \mathfrak{x}_{\hat{p}}, SP[k_1], SP[k_2], XP[k_1], XP[k_2]$: $\mathsf{ComVfy}(\mathfrak{C}_{\hat{p}}, \mathfrak{x}_{\hat{p}}, SP[k_1] \cdot SP[k_2]) \wedge \mathsf{ComVfy}(CP[k_1], XP[k_1], SP[k_1]) \wedge \mathsf{ComVfy}(CP[k_2], XP[k_2], SP[k_2])$. The value $\mathfrak{C}_{\hat{p}}$ is delivered to $\mathcal{Q}$ via $\mathcal{F}_{\text{gZK}}$.	
$\mathcal{Q}$ proves the following to $\mathcal{P}$ using $\mathcal{F}_{\text{gZK}}$ with label $\langle\mathtt{m6},\varphi\rangle$: $\exists \mathfrak{x}_{\hat{q}}, SQ[k_1], SQ[k_2], XQ[k_1], XQ[k_2]$: $\mathsf{ComVfy}(\mathfrak{C}_{\hat{q}}, \mathfrak{x}_{\hat{q}}, SQ[k_1] \cdot SQ[k_2]) \wedge \mathsf{ComVfy}(CQ[k_1], XQ[k_1], SQ[k_1]) \wedge \mathsf{ComVfy}(CQ[k_2], XQ[k_2], SQ[k_2])$. The value $\mathfrak{C}_{\hat{q}}$ is delivered to $\mathcal{P}$ via $\mathcal{F}_{\text{gZK}}$.	
Run Π_{mul} with $\mathcal{Q}$ with input $(\mathtt{P}, SP[k_1], CP[k_1], XP[k_1], CQ[k_2], \langle\mathtt{m7},\varphi\rangle)$ and get $(\tilde{u}, \mathfrak{C}_{\tilde{u}}, \mathfrak{x}_{\tilde{u}}, \mathfrak{C}_{\tilde{v}})$ as output.	Run Π_{mul} with $\mathcal{P}$ with input $(\mathtt{Q}, SQ[k_2], CQ[k_2], XQ[k_2], CP[k_1], \langle\mathtt{m7},\varphi\rangle)$ and get $(\tilde{v}, \mathfrak{C}_{\tilde{v}}, \mathfrak{x}_{\tilde{v}}, \mathfrak{C}_{\tilde{u}})$ as output.
Run Π_{mul} with $\mathcal{Q}$ with input $(\mathtt{P}, SP[k_2], CP[k_2], XP[k_2], CQ[k_1], \langle\mathtt{m8},\varphi\rangle)$ and get $(u, \mathfrak{C}_u, \mathfrak{x}_u, \mathfrak{C}_v)$ as output.	Run Π_{mul} with $\mathcal{Q}$ with input $(\mathtt{Q}, SQ[k_1], CQ[k_1], XQ[k_1], CP[k_2], \langle\mathtt{m8},\varphi\rangle)$ and get $(v, \mathfrak{C}_v, \mathfrak{x}_v, \mathfrak{C}_u)$ as output.
Wait until all four threads are done before proceeding.	
Compute own share: $SP[k_0] \leftarrow \hat{p} + \tilde{u} + u$; commitment: $CP[k_0] \leftarrow \mathfrak{C}_{\hat{p}} * \mathfrak{C}_{\tilde{u}} * \mathfrak{C}_u$; opening: $XP[k_0] \leftarrow \mathfrak{x}_{\hat{p}} + \mathfrak{x}_{\tilde{u}} + \mathfrak{x}_u$. $\mathcal{Q}$'s commitment: $CQ[k_0] \leftarrow \mathfrak{C}_{\hat{q}} * \mathfrak{C}_{\tilde{v}} * \mathfrak{C}_v$. Mark value as ready: $RP \leftarrow k_0$.	Compute own share: $SQ[k_0] \leftarrow \hat{q} + \tilde{v} + v$; commitment: $CQ[k_0] \leftarrow \mathfrak{C}_{\hat{q}} * \mathfrak{C}_{\tilde{v}} * \mathfrak{C}_v$; opening: $XQ[k_0] \leftarrow \mathfrak{x}_{\hat{q}} + \mathfrak{x}_{\tilde{v}} + \mathfrak{x}_v$. $\mathcal{P}$'s commitment: $CP[k_0] \leftarrow \mathfrak{C}_{\hat{p}} * \mathfrak{C}_{\tilde{u}} * \mathfrak{C}_u$. Mark value as ready: $RQ \leftarrow k_0$.

Fig. 4: Multiplication. The subroutine Π_{mul} is defined in Section 4.3 and Figure 6.

4.2 Realizing Π_{gABB}

The Π_{gABB} protocol (the realization of $\mathcal{F}_{\text{gABB}}$) adds the possibility of doing a proof of *existence* of external variables inside the *Proof* functionality. In the *Proof* functionality of Π_{gABB}, $\mathcal{P}$ may now specify a non-empty list w_e of witnesses whose *existence* is proven. See Figure 5.

4.3 The Π_{mul} Subroutine for Multiplication of Committed Inputs

We now give the construction of the 2-party $\mathcal{F}_{\text{gZK}}$-hybrid protocol Π_{mul} for multiplication of committed inputs, which we use as a subroutine in Π_{ABB} in the multiplication instruction. In a nutshell: on $\mathcal{P}$'s private input a and $\mathcal{Q}$'s private input b, Π_{mul} outputs shares to the product: u to $\mathcal{P}$ and v to $\mathcal{Q}$, such that $u + v = a \cdot b$.

The protocol draws on ideas from Ishai et al's $\tilde{\pi}^{\mathsf{OT}}$ protocol—defined in Appendix A.2 of the full version of their paper [IPS08]—and uses a similar approach as many two-party computation protocols (e.g., Damgård and Orlandi's π_{mul} protocol [DO10b]). We fleshed out the details of Ishai et al.'s protocol to make it secure against *active* adversaries, improve its efficiency, and integrate it into our overall protocol.

The basic idea of the protocol is for $\mathcal{P}$ and $\mathcal{Q}$ to first obtain shares y and $(-t)$ on the product of two *random* values w and s, respectively: $y - t = w \cdot s$; second to erase all intermediate state used in the previous step; third to exchange the values $\sigma = (a - w)$ and $\delta = (b - s)$; and finally to obtain shares on the product of the actual input values a and b by outputting $u = \delta \cdot a + y$

$\mathcal{P}$ **proceeds as follows**:	$\mathcal{Q}$ **proceeds as follows**:
$\mathcal{P}$'s input is $\langle \varphi, x, \tilde{m}, w_k, \hat{m}, w_e \rangle$. She parses φ like $\mathcal{Q}$; x is an instance for R; $\tilde{m} \in \mathbb{N}$; $\hat{m} \in \mathbb{N}$; $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1} \rangle$ and $w_e = \langle w_{e,0}, \ldots, w_{e,\hat{m}-1} \rangle$ are lists of witnesses. For Π_{ABB}, we require that $\hat{m} = 0$, i.e., w_e is an empty list. When constructing Π_{gABB}, the realization of $\mathcal{F}_{\text{gABB}}$, w_e may be non-empty.	$\mathcal{Q}$'s input is $\langle \varphi \rangle$, where $\varphi = \langle m, \langle k_0, \ldots, k_{m-1} \rangle, R \rangle$; R is a predicate that is compatible with $\mathcal{F}_{\text{gZK}}$; $m \in \mathbb{N}$; and $\forall i \in \mathbb{N}_m : k_i \in \Sigma^\star$.
Wait until $\forall i \in \mathbb{N}_m : k_i \in KP$.	Wait until $\forall i \in \mathbb{N}_m : k_i \in KP$.
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\text{gZK}}$ parametrized with R and with label $\langle \mathsf{pp}, \varphi \rangle$: $\mathsf{K} w_{k,0}, \ldots, w_{k,\tilde{m}-1}\ \exists w_{e,0}, \ldots, w_{e,\hat{m}-1}, V[k_0], \ldots, V[k_{m-1}], XP[k_0], \ldots, XP[k_{m-1}] :$ $\bigwedge_{i=0}^{m-1} \mathsf{ComVfy}(CP[k_i], XP[k_i], V[k_i] - SQ[k_i]) \ \wedge$ $R\big(x, (w_{k,0}, \ldots, w_{k,\tilde{m}-1}) \cup (w_{e,0}, \ldots, w_{e,\hat{m}-1}) \cup (V[k_i], \ldots, V[k_{m-1}])\big) = 1$. The instance of the statement to be proven, x, is delivered to $\mathcal{Q}$ via $\mathcal{F}_{\text{gZK}}$. $\mathcal{P}$ erases her private inputs before delivering x.	
	$\mathcal{Q}$ **returns** x.

Fig. 5: Proof by $\mathcal{P}$.

and $v = \sigma \cdot b - t$, respectively. Commitments and relevant proofs are used during all steps. We refer to Figure 6 for the construction.

The erasure in Step 2 is needed to ensure security against *adaptive* adversaries: since the encryption scheme used in our protocol is not receiver–non-committing [CHK05], the simulator cannot produce a convincing view of the first step for any other value of w. In fact, there are no known practical receiver–non-committing schemes that satisfy our requirements. By erasing state in Step 2, the simulator is dispensed with producing that view in Step 3.

4.4 Efficiency Considerations for the Zero-Knowledge Proofs in Π_{ABB}

Careful design enables us to achieve a very efficient and practical construction. In particular, we minimize the amount of computation required inside the realization π of the zero-knowledge proof functionality $\mathcal{F}_{\text{gZK}}$, which accounts for the majority of the runtime of our protocol, as follows.

1) Instead of using the Paillier encryption scheme as in Camenisch et al. [CKS11] to verifiably encrypt the witnesses whose *knowledge* is proven in π, we use the Camenisch-Shoup encryption scheme with short keys, short randomness, and with modulus n^2. Paillier encryption implies the use of a different modulus, since the simulator needs to know its factorization to extract the witnesses.

2) We use homomorphic commitment and encryption schemes that work with groups of the same order n. Most of the witnesses used in $\mathcal{F}_{\text{gZK}}$ therefore live in a group of known order n, and most operations inside π stay inside groups of order n. We therefore do not need to encrypt values larger than n in π, and can avoid expensive integer commitments in π [CKS11].

3) We use the cheaper proofs of *existence* [CKS11] instead of proofs of *knowledge* wherever possible. This reduces the number of verifiable encryptions needed inside π.

4) Finally, we use an encryption scheme in Π_{mul} where the proof of correctness of key generation is cheap. (For Camenisch-Shoup encryption with full key length and Paillier encryption, this proof is very expensive.)

$\mathcal{P}$ **proceeds as follows**:	$\mathcal{Q}$ **proceeds as follows**:
$\mathcal{P}$'s input is $(\mathtt{P}, a, \mathfrak{C}_a, \mathfrak{x}_a, \mathfrak{C}_b, \lambda)$.	$\mathcal{Q}$'s input is $(\mathtt{Q}, b, \mathfrak{C}_b, \mathfrak{x}_b, \mathfrak{C}_a, \lambda)$.
$(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(\mathsf{n})$; $\quad w \xleftarrow{\$} \mathbb{Z}_{\mathsf{n}}$; $(\mathsf{E}_w, \mathsf{r}_w) \xleftarrow{\$} \mathsf{Enc}(w, \mathsf{pk})$.	$s \xleftarrow{\$} \mathbb{Z}_{\mathsf{n}}$; $\quad t \xleftarrow{\$} \mathbb{Z}_{\mathsf{n}}$; $(\mathfrak{C}_s, \mathfrak{x}_s) \xleftarrow{\$} \mathsf{Com}(s)$; $\quad (\mathfrak{C}_t, \mathfrak{x}_t) \xleftarrow{\$} \mathsf{Com}(t)$.
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\mathrm{gZK}}$ with label $\langle \mathtt{cm1}, \lambda \rangle$: $\mathsf{K} w \; \exists \mathsf{sk} : (\mathsf{pk}, \mathsf{sk}) \in \mathsf{KeyGen}(\mathsf{n}) \wedge w = \mathsf{Dec}(\mathsf{E}_w, \mathsf{sk})$. The values E_w, pk are delivered to $\mathcal{Q}$ via $\mathcal{F}_{\mathrm{gZK}}$ after $\mathcal{P}$ securely erases r_w.	
$\sigma \leftarrow a - w$.	$(\mathsf{E}_t, \mathsf{r}_t) \xleftarrow{\$} \mathsf{Enc}(t, \mathsf{pk})$; $\quad \mathsf{E}_y \leftarrow (\mathsf{E}_w)^s * \mathsf{E}_t$.
$\mathcal{Q}$ proves the following to $\mathcal{P}$ using $\mathcal{F}_{\mathrm{gZK}}$ with label $\langle \mathtt{cm2}, \lambda \rangle$: $\mathsf{K} s \; \exists t, \mathfrak{x}_s, \mathfrak{x}_t, \mathsf{r}_t : \mathsf{ComVfy}(\mathfrak{C}_s, \mathfrak{x}_s, s) \wedge \mathsf{ComVfy}(\mathfrak{C}_t, \mathfrak{x}_t, t) \wedge \mathsf{E}_y = (\mathsf{E}_w)^s * \mathsf{Enc}(t, \mathsf{pk}, \mathsf{r}_t)$. The values $\mathfrak{C}_s$, $\mathfrak{C}_t$ and E_y are delivered to $\mathcal{P}$ via $\mathcal{F}_{\mathrm{gZK}}$ after $\mathcal{Q}$ securely erases r_t.	
$y \leftarrow \mathsf{Dec}(\mathsf{E}_y, \mathsf{sk})$; $\quad (\mathfrak{C}_y, \mathfrak{x}_y) \xleftarrow{\$} \mathsf{Com}(y)$.	$\delta \leftarrow b - s$; $\quad \mathfrak{x}_\delta \leftarrow \mathfrak{x}_b - \mathfrak{x}_s$.
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\mathrm{gZK}}$ with label $\langle \mathtt{cm3}, \lambda \rangle$: $\exists y, w, \mathfrak{x}_y, \mathfrak{x}_a, \mathsf{sk} : \mathsf{ComVfy}(\mathfrak{C}_y, \mathfrak{x}_y, y) \wedge y = \mathsf{Dec}(\mathsf{E}_y, \mathsf{sk}) \wedge w = \mathsf{Dec}(\mathsf{E}_w, \mathsf{sk}) \wedge$ $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{KeyGen}(\mathsf{n}) \wedge \mathsf{ComVfy}(\mathfrak{C}_a, \mathfrak{x}_a, w + \sigma)$. The values $\mathfrak{C}_y$, σ are delivered to $\mathcal{Q}$ via $\mathcal{F}_{\mathrm{gZK}}$ after $\mathcal{P}$ securely erases sk.	
$\mathcal{Q}$ proves the following to $\mathcal{P}$ using $\mathcal{F}_{\mathrm{gZK}}$ with label $\langle \mathtt{cm4}, \lambda \rangle$: $\exists \mathfrak{x}_\delta : \mathsf{ComVfy}(\mathfrak{C}_b * (\mathfrak{C}_s)^{-1}, \mathfrak{x}_\delta, \delta)$. The value δ is delivered to $\mathcal{P}$ via $\mathcal{F}_{\mathrm{gZK}}$.	
Compute own share: $u \leftarrow \delta \cdot a + y$; opening: $\mathfrak{x}_u \leftarrow \mathfrak{x}_a \cdot \delta + \mathfrak{x}_y$; and commitment: $\mathfrak{C}_u \leftarrow (\mathfrak{C}_a)^\delta * \mathfrak{C}_y$. Compute $\mathcal{Q}$'s commitment: $\mathfrak{C}_v \leftarrow (\mathfrak{C}_s)^\sigma * (\mathfrak{C}_t)^{-1}$. $\mathcal{P}$ **returns** $(u, \mathfrak{C}_u, \mathfrak{x}_u, \mathfrak{C}_v)$.	Compute own share: $v \leftarrow \sigma \cdot s - t$; opening: $\mathfrak{x}_v \leftarrow \mathfrak{x}_s \cdot \sigma - \mathfrak{x}_t$; and commitment: $\mathfrak{C}_v \leftarrow (\mathfrak{C}_s)^\sigma * (\mathfrak{C}_t)^{-1}$. Compute $\mathcal{P}$'s commitment: $\mathfrak{C}_u \leftarrow (\mathfrak{C}_a)^\delta * \mathfrak{C}_y$. $\mathcal{Q}$ **returns** $(v, \mathfrak{C}_v, \mathfrak{x}_v, \mathfrak{C}_u)$.

Fig. 6: The Π_{mul} protocol.

5 Additional Instructions for $\mathcal{F}_{\mathrm{ABB}}$

We will start this section by showing how one can create a $\mathcal{F}_{\mathrm{ABB}}$-hybrid protocol that includes additional instructions for generating random numbers, random bits, inverting, and doing several other useful operations. Certain useful instructions however require that the $\mathcal{F}_{\mathrm{ABB}}$ functionality itself is modified and not just used as a building block: we show here a new output instruction for $\mathcal{F}_{\mathrm{ABB}}$ that returns a value exponentiated by a certain group element g instead of revealing the value directly; we will use that instruction in Section 8 for constructing an oblivious pseudorandom function that is secure against *dynamic* corruptions in the UC model.

5.1 Instructions as Part of a Higher-Level Protocol

Random integers. A random value can be shared as follows: $\mathcal{P}$ and $\mathcal{Q}$ each choose a random number $a \xleftarrow{\$} \mathbb{Z}_{\mathsf{n}}$ and $b \xleftarrow{\$} \mathbb{Z}_{\mathsf{n}}$, respectively, input it into $\mathcal{F}_{\mathrm{ABB}}$, and finally sum their inputs $c \leftarrow a + b$ using the *Linear Combination* instruction. Provided that at least one of the two is honest, the value c is uniformly distributed in $\mathbb{Z}_{\mathsf{n}}$.

Random bits. $\mathcal{P}$ and $\mathcal{Q}$ can share a random bit as follows: $\mathcal{P}$ and $\mathcal{Q}$ each choose a random number $a \xleftarrow{\$} \{-1, 1\}$ and $b \xleftarrow{\$} \{-1, 1\}$, respectively, and input it to $\mathcal{F}_{\mathrm{ABB}}$. They then compute a^2 and b^2 using the *Multiplication* instruction, and reveal the result to each other. The protocol aborts if $a^2 \neq 1$ or $b^2 \neq 1$. They then compute $c \leftarrow a \cdot b$. The value c is now uniformly distributed

in $\{-1, 1\}$, provided that at least one of the two parties is honest and the factorization of n is unknown to both of them. To adjust the random value to $\mathbb{Z}_2$, they can compute $d \leftarrow c \cdot (1/2 \pmod{\mathsf{n}}) + (1/2 \pmod{\mathsf{n}})$.

Inversion. This algorithm is based on a technique by Bar-Ilan and Beaver [BIB89]:

1. Let $V[k_1]$ denote the value to invert.
2. $\mathcal{P}$ and $\mathcal{Q}$ choose a random integer $V[k_2]$ as shown earlier in this section.
3. They multiply both values: $V[k_3] \leftarrow V[k_1] \cdot V[k_2]$.
4. The product is output first to $\mathcal{P}$, and then to $\mathcal{Q}$: $v_3 \leftarrow V[k_3]$.
5. They invert the value: $v_4 \leftarrow v_3^{-1} \pmod{\mathsf{n}}$ (abort if v_3 is not invertible.);
6. and compute the result: $V[k_5] \leftarrow V[k_2] \cdot v_4 = V[k_2] \cdot (V[k_1] \cdot V[k_2])^{-1} = (V[k_1])^{-1}$.

As long as $V[k_1]$ is invertible mod n, this protocol aborts with negligible probability, is correct, and perfectly preserves the privacy of $V[k_1]$. If $V[k_1] = 0$, this fact will be revealed. As we assumed the factorization of n to be unknown, we can safely ignore the case where $V[k_1]$ is a multiple of a non-trivial factor of n.

Other useful operations. Our protocol is almost compatible with the algorithms by Damgård, Fitzi et al. [DFK+06] for performing comparisons (including inequalities), bit decompositions, modular reduction, modular exponentiation, etc. of the values in the circuit. Their setting assumed that the values in the circuit are in a prime order group, but in our scheme n is composite. Fortunately the only operation that they use in their paper that cannot be performed in a composite order group—finding a square root modulo n—is needed only for generating random bits; by replacing that algorithm by the version presented earlier, no more problems remain.

5.2 Modifying $\mathcal{F}_{\text{ABB}}$ to Add New Instructions

Unfortunately there are some useful instructions that cannot be added "on top of" $\mathcal{F}_{\text{ABB}}$ as described in the previous subsection, but have to be included "inside" $\mathcal{F}_{\text{ABB}}$: the UC composition theorem can therefore not be applied, and the security proof has to be redone. We give here an example of such an instruction: it is a variant of the output instruction that outputs not $V[k]$ but $\mathsf{g}^{V[k]}$, where $\langle \mathsf{g} \rangle = \mathbb{G}$ is some abelian group (written multiplicatively) of order n.

Informal definition of the ideal functionality. The high-level description of the additional instructions is the following:

- *Exponentiated output to $\mathcal{P}$:* $\mathcal{F}_{\text{ABB}}$ parses the common input φ as $\langle k, \mathbb{G}, \mathsf{g} \rangle$ where $\mathbb{G}$ is the description of some group (written multiplicatively) of order n, and $\mathsf{g} \in \mathbb{G}$ is a generator of $\mathbb{G}$. $\mathcal{F}_{\text{ABB}}$ delivers $\mathsf{g}^{V[k]}$ to $\mathcal{P}$.
- *Exponentiated output to $\mathcal{Q}$:* Idem, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

Formal definition of the ideal functionality. The formal definition of the additional instructions is the following:

- **Exponentiated output to $\mathcal{P}$:** In this instruction, parse φ as $\langle k, \mathbb{G}, \mathsf{g} \rangle$ where $k \in \Sigma^\star$, $\mathbb{G}$ is the description of some group (written multiplicatively) of order n, and $\mathsf{g} \in \mathbb{G}$ is a generator of $\mathbb{G}$.
  - $\mathsf{ep{:}p{:}}\varphi$: Accept message $\langle \mathsf{ep{:}p{:}}\varphi \rangle$ from $\mathcal{P}$, where the identifier is ready to be used: $k \in RP$. Send $\langle \mathsf{ep{:}p{:}}\varphi \rangle$ to $\mathcal{A}$.

$\mathcal{P}$ **proceeds as follows**:	$\mathcal{Q}$ **proceeds as follows**:
Both parties' input is (φ). It is parsed as $\varphi = \langle k, \mathbb{G}, \mathsf{g}\rangle$ with $k \in \Sigma^\star$, $\mathbb{G}$ the description of a group of order n, and g a generator of $\mathbb{G}$.	
Wait until $k \in RP$. Exponentiate share: $v \leftarrow \mathsf{g}^{SP[k]}$.	Wait until $k \in RQ$.
$\mathcal{P}$ proves the following to $\mathcal{Q}$ using $\mathcal{F}_{\text{gZK}}$ with label $\langle \mathtt{eq}, \varphi\rangle$: $\exists XP[k], SP[k] : \mathsf{ComVfy}(CP[k], XP[k], SP[k]) \wedge v = \mathsf{g}^{SP[k]}$. The value v is delivered to $\mathcal{Q}$ via $\mathcal{F}_{\text{gZK}}$.	
	$\mathcal{Q}$ **returns** $\mathsf{g}^{SQ[k]} \cdot v$.

Fig. 7: Exponentiated output to $\mathcal{Q}$.

- $\mathtt{ep{:}q}{:}\varphi$: Accept message $\langle \mathtt{ep{:}q}{:}\varphi\rangle$ from $\mathcal{Q}$, where the identifier is ready to be used: $k \in RQ$. Send $\langle \mathtt{ep{:}q}{:}\varphi\rangle$ to $\mathcal{A}$.
- $\mathtt{ep{:}lock}{:}\varphi$ $[\mathtt{ep{:}p}{:}\varphi \wedge \mathtt{ep{:}q}{:}\varphi]$: Accept $\langle \mathtt{ep{:}lock}{:}\varphi\rangle$ from $\mathcal{A}$. Send $\langle\rangle$ to $\mathcal{A}$.
- $\mathtt{ep{:}deliver}{:}\varphi$ $[\mathtt{ep{:}lock}{:}\varphi]$: Accept $\langle \mathtt{ep{:}deliver}{:}\varphi\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{ep{:}deliver}{:}\varphi, \mathsf{g}^{V[k]}\rangle$ to $\mathcal{P}$.
- $\mathtt{ep{:}done}{:}\varphi$ $[\mathtt{ep{:}lock}{:}\varphi]$: Accept $\langle \mathtt{ep{:}done}{:}\varphi\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{ep{:}done}{:}\varphi\rangle$ to $\mathcal{Q}$.

- **Exponentiated output to $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to $\mathtt{eq}$. We do not formalize this instruction here.

Construction of exponentiated output to $\mathcal{Q}$. $\mathcal{Q}$ retrieves an exponentiated value $\mathsf{g}^{V[k]}$, where $\mathbb{G}$ and $\mathsf{g} \in \mathbb{G}$ can be chosen freely. Concretely, $\mathcal{P}$ exponentiates her share and sends it to $\mathcal{Q}$ together with a proof of correctness. See Figure 7 for the construction.

6 Security Proof

In this section we start with a description of the main ideas of the security proof. The proof proceeds in two steps: we first prove that our protocol is secure when run with *nice* environments. We then apply the special composition theorem of Camenisch et al. [CKS11] to prove that our protocol is secure against *all* environments.

6.1 Main Ideas

We use the standard approach for proving the security of protocols in the UC or GNUC models: we construct a straight-line simulator $\mathcal{S}$ such that for all polynomial-time–bounded environments $\mathcal{Z}$ and all polynomial-time–bounded adversaries $\mathcal{A}$, the environment $\mathcal{Z}$ cannot distinguish a protocol execution with $\mathcal{A}$ and Π_{ABB} in the $(\mathcal{F}_{\text{ach}}, \mathcal{F}_{\text{gZK}})$-hybrid "real" world from a protocol execution with $\mathcal{S}$ and $\mathcal{F}_{\text{ABB}}$ in the "ideal" world. We prove that $\mathcal{Z}$ cannot distinguish these two worlds by defining a sequence of intermediate "hybrid" worlds (the first one being the real world and the last one the ideal world) and showing that $\mathcal{Z}$ cannot distinguish between any two consecutive hybrid worlds in that sequence. We follow the formalism of the GNUC framework to deal with CRS's and system parameters (see Section 10 of the GNUC paper [HS11]).

The main difficulties in constructing the simulator $\mathcal{S}$ are as follows: *1)* $\mathcal{S}$ has to extract the inputs of all corrupted parties; *2)* $\mathcal{S}$ has to compute and send commitments and ciphertexts on

behalf of the honest parties without knowing their inputs, i.e., $\mathcal{S}$ cannot commit and encrypt the right values; *3)* when an honest party gets corrupted mid-protocol, $\mathcal{S}$ has to provide to $\mathcal{A}$ the full *non-erased* intermediate state of the party, in particular the opening of the commitments and the randomness of the encryptions.

To address the first difficulty, recall that the parties are required to perform a proof of *knowledge* of all new inputs to the circuit. The simulator $\mathcal{S}$ can therefore recover the input of all corrupted parties with the help of $\mathcal{F}_{\text{gZK}}$. In the first few hybrid worlds, the statistically binding commitments ensure that the values in the circuit stay consistent with the inputs. In the subsequent hybrid worlds, the computational indistinguishability of the two types of CRS ensure that the adversary cannot equivocate commitments even when $\mathcal{S}$ uses the perfectly-hiding CRS with trapdoor.

We now address the second and third difficulty. Upon corruption of a party, $\mathcal{S}$ is allowed to recover the original input of that party from $\mathcal{F}_{\text{ABB}}$. By using the perfectly-hiding CRS with trapdoor, $\mathcal{S}$ can equivocate all commitments it made so far to ensure that the committed values are consistent with the view of the adversary. By construction, $\mathcal{S}$ never needs to reveal the randomness used for an encryption for which it does not know the plaintext. Recall that in Π_{mul}, the parties first encrypt a random offset, then erase the decryption key and the randomness used to encrypt, and only then deliver the encryption of the offset plus party's input to the adversary (recall that $\mathcal{F}_{\text{gZK}}$ allows the erasure of witnesses *before* delivering the statement to be proven to the other party). The simulator $\mathcal{S}$ can adjust the offset so that the view delivered to the adversary is consistent. See also Appendix A.2 of Ishai et al.'s paper [IPS08].

The rest of the security proof is now straightforward.

6.2 Security Proof

Let $\Pi_{\text{ABB}}^{\pi/\mathcal{F}_{\text{gZK}}}$ be the $(\mathcal{F}_{\text{sch}}, \mathcal{F}_{\text{ach}})$-hybrid protocol in which every instance of $\mathcal{F}_{\text{gZK}}$ in Π_{ABB} has been replaced by the zero-knowledge protocol described in Camenisch et al.'s paper [CKS11]. To prove our scheme secure, we need to prove the following theorem:

Theorem 1. *There exists a simulator $\mathcal{S}$, such that for all polynomial-time-bounded environments $\mathcal{Z}$ and all polynomial-time-bounded adversaries $\mathcal{A}$:*

$$\text{Exec}(\Pi_{ABB}^{\pi/\mathcal{F}_{gZK}}, \mathcal{A}, \mathcal{Z}) \approx \text{Exec}(\mathcal{F}_{ABB}, \mathcal{S}, \mathcal{Z}).$$

In the theorem above, $\text{Exec}(\Pi_{\text{ABB}}^{\pi/\mathcal{F}_{\text{gZK}}}, \mathcal{A}, \mathcal{Z})$ denotes the binary random variable given by the output of $\mathcal{Z}$ when interacting with $\mathcal{A}$ and $\Pi_{\text{ABB}}^{\pi/\mathcal{F}_{\text{gZK}}}$ in the $(\mathcal{F}_{\text{ach}}, \mathcal{F}_{\text{gZK}})$-hybrid world, and analogously for $\text{Exec}(\mathcal{F}_{\text{ABB}}, \mathcal{S}, \mathcal{Z})$ in the *ideal world*. The symbol $\approx$ means statistically close.

To prove the theorm, we first need to prove the following lemma:

Lemma 1. *There exists a simulator $\mathcal{S}$ that does not extract the witnesses quantified by $\exists$ in any $\mathcal{F}_{gZK}$, such that for all polynomial-time-bounded* nice *environments $\mathcal{Z}$ and all polynomial-time-bounded adversaries $\mathcal{A}$:*

$$\text{Exec}(\Pi_{ABB}, \mathcal{A}, \mathcal{Z}) \approx \text{Exec}(\mathcal{F}_{ABB}, \mathcal{S}, \mathcal{Z}).$$

A *nice* environment is an environment that never asks $\mathcal{A}$ to submit a false statement to $\mathcal{F}_{\text{gZK}}$ [CKS11].

Proof of Lemma 1. In Section 6.3, we construct a simulator $\mathcal{S}$ and prove that is satisfies the requirements of the Lemma 1.

Proof of Theorem 1. Since the simulator we constructed in Section 6.3 satisfies the requirements of Lemma 1, we can apply the special composition theorem of Camenisch et al. [CKS11], to conclude that the simulator also satisfies the requirements of Theorem 1.

Conclusion. From Theorem 1, we can conclude that the $(\mathcal{F}_{\text{sch}}, \mathcal{F}_{\text{ach}})$-hybrid protocol $\Pi_{\text{ABB}}^{\pi/\mathcal{F}_{\text{gZK}}}$ is a secure realization of the ideal functionality $\mathcal{F}_{\text{ABB}}$, and is universally composable. This concludes the security proof.

6.3 Proof of Lemma 1

Notation. We adopt the convention that the ideal functionalities in the $(\mathcal{F}_{\text{gZK}}, \mathcal{F}_{\text{ach}})$-hybrid "real" world (and which are controlled by $\mathcal{S}$) are surrounded by quotes: "$\mathcal{F}_{\text{gZK}}$" and "$\mathcal{F}_{\text{ach}}$". Note that $\mathcal{S}$ does not have to run these ideal functionalities honestly, it just needs to ensure that the messages $\mathcal{S}$ sends on their behalf are indistinguishable from an honest execution. Furthermore, we denote the parties in the real world as "$\mathcal{P}$" and "$\mathcal{Q}$". When such a party is honest, it is controlled by $\mathcal{S}$; when that party is corrupted, it is controlled by the adversary $\mathcal{A}$.

The simulator $\mathcal{S}$ is a six-interface system. The simulator $\mathcal{S}$ communicates with $\mathcal{F}_{\text{ABB}}$ through 3 interfaces: the $\mathcal{S}$-interface (where $\mathcal{F}_{\text{ABB}}$ sends data to and receives data from the ideal adversary), the $\mathcal{P}$-interface (which is active only after $\mathcal{P}$ becomes corrupted, and where $\mathcal{F}_{\text{ABB}}$ sends data to and receives data from the *corrupted* $\mathcal{P}$) and the $\mathcal{Q}$-interface (idem but for $\mathcal{Q}$). The simulator $\mathcal{S}$ runs one instance of the real-world adversary $\mathcal{A}$. It relays all messages between $\mathcal{Z}$ and $\mathcal{A}$. The simulator $\mathcal{S}$ communicates with $\mathcal{A}$ through 3 interfaces: the "$\mathcal{A}$"-interface (connected to the adversary interfaces of all "$\mathcal{F}_{\text{gZK}}$" and "$\mathcal{F}_{\text{ach}}$" used in the protocol execution), the "$\mathcal{P}$"-interface (which is active only after $\mathcal{P}$ becomes corrupted, and which is connected to the $\mathcal{P}$-interface of all "$\mathcal{F}_{\text{gZK}}$" and "$\mathcal{F}_{\text{ach}}$" used in the protocol execution), and the "$\mathcal{Q}$" interface (idem for $\mathcal{Q}$). See Figures 8, 9, 10, and 11 for a schematic representation of the construction of $\mathcal{S}$ in the cases where no parties are corrupted, $\mathcal{P}$ is corrupted, $\mathcal{Q}$ is corrupted, and all parties are corrupted, respectively.

Initialization. Before running $\mathcal{A}$ for the first time, $\mathcal{S}$ programs the CRS using ComGen'_0 so that commitments are perfectly hiding, and so that $\mathcal{S}$ knows the trapdoor t which will enable it to equivocate all commitments it makes on behalf of "$\mathcal{P}$" and "$\mathcal{Q}$".

Since n is part of the system parameters, $\mathcal{S}$ does not know its factorization.

$\mathcal{P}$ and $\mathcal{Q}$ honest. When $\mathcal{P}$ and $\mathcal{Q}$ are both honest, $\mathcal{A}$ sees only status messages without any content. The construction of $\mathcal{S}$ is therefore straightforward. See Figure 8. For completeness, we will show the behaviour of $\mathcal{S}$ for the *Input from* $\mathcal{P}$ and *Multiplication* instructions. The behaviour of $\mathcal{S}$ for all other instructions is similar to its behaviour for *Input from* $\mathcal{P}$.

Input from $\mathcal{P}$. Upon receiving $\langle \texttt{ip:send:}\varphi \rangle$ from $\mathcal{F}_{\text{ABB}}$ (through the $\mathcal{S}$-interface), send $\langle \texttt{send}, \ell(\ldots) \rangle$ to $\mathcal{A}$ (through the "$\mathcal{A}$"-interface). The length of the statement and witnesses is fixed, so $\mathcal{S}$ knows what value $\ell(\ldots)$ to send.

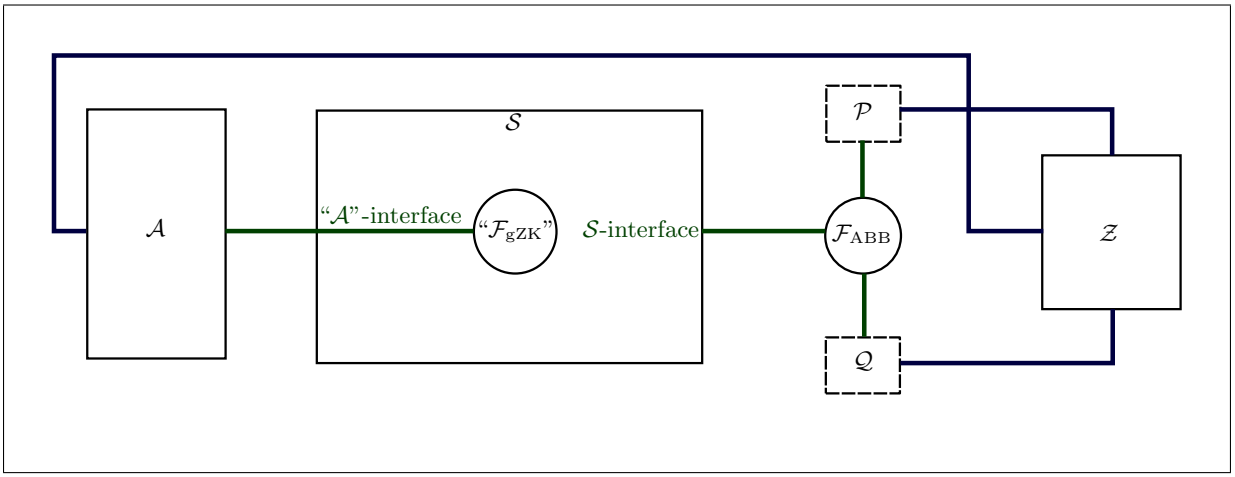

Fig. 8: Construction of $\mathcal{S}$ in case all parties are honest. For simplicity, we chose to represent only one ideal functionality in the construction of $\mathcal{S}$.

Upon receiving $\langle \texttt{ip:ready}{:}\varphi \rangle$ from $\mathcal{F}_{\text{ABB}}$, send $\langle \texttt{ready} \rangle$ to $\mathcal{A}$.

Upon receiving $\langle \texttt{lock} \rangle$ from $\mathcal{A}$, send $\langle \texttt{ip:lock}{:}\varphi \rangle$ to $\mathcal{F}_{\text{ABB}}$. Wait for $\langle\rangle$ from $\mathcal{F}_{\text{ABB}}$, and send $\langle\rangle$ to $\mathcal{A}$.

Upon receiving $\langle \texttt{deliver} \rangle$ from $\mathcal{A}$, send $\langle \texttt{ip:deliver}{:}\varphi \rangle$ to $\mathcal{F}_{\text{ABB}}$.

Upon receiving $\langle \texttt{done} \rangle$ from $\mathcal{A}$, send $\langle \texttt{ip:done}{:}\varphi \rangle$ to $\mathcal{F}_{\text{ABB}}$.

Multiplication. This instruction is more complex than all others, since it contains several independant instances of "$\mathcal{F}_{\text{gZK}}$". We divide the "$\mathcal{A}$"-interface into m sub-interfaces (numbered from 1 to m), one for each instance of "$\mathcal{F}_{\text{gZK}}$". To simplify the discussion, we only consider the "single-thread" case, i.e., we assume the four "threads" in the multiplication instruction of Π_{ABB} run sequentially, one after the other.

Upon receiving $\langle \texttt{m:p}{:}\varphi \rangle$ from $\mathcal{F}_{\text{ABB}}$, send $\langle \texttt{send}, \ell(\ldots) \rangle$ to $\mathcal{A}$ via the *first* sub-interface (of the "$\mathcal{A}$"-interface). The length of the statement and witness is fixed, so $\mathcal{S}$ knows what value $\ell(\ldots)$ to send.

Upon receiving $\langle \texttt{m:q}{:}\varphi \rangle$ from $\mathcal{F}_{\text{ABB}}$, send $\langle \texttt{ready} \rangle$ to $\mathcal{A}$ via the *first* sub-interface.

Upon receiving $\langle \texttt{lock} \rangle$ from $\mathcal{A}$ via the ith sub-interface, where $i \neq m$, send $\langle\rangle$ to $\mathcal{A}$ via the ith sub-interface.

Upon receiving $\langle \texttt{lock} \rangle$ from $\mathcal{A}$ via the mth sub-interface, send $\langle \texttt{m:lock}{:}\varphi \rangle$ to $\mathcal{F}_{\text{ABB}}$. Wait for $\langle\rangle$ from $\mathcal{F}_{\text{ABB}}$, and send $\langle\rangle$ to $\mathcal{A}$ via the mth sub-interface.

Upon receiving $\langle \texttt{done} \rangle$ from $\mathcal{A}$ via the ith sub-interface, where $i \neq m$, send $\langle \texttt{ready} \rangle$ to $\mathcal{A}$ via the $(i+1)$st sub-interface.

Upon receiving $\langle \texttt{deliver} \rangle$ from $\mathcal{A}$ via the ith sub-interface, where $i \neq m$, send $\langle \texttt{send}, \ell(\ldots) \rangle$ to $\mathcal{A}$ via the $(i+1)$st sub-interface. The length $\ell(\ldots)$ is easy for $\mathcal{S}$ to determine.

Upon receiving $\langle \texttt{done} \rangle$ from $\mathcal{A}$ via the mth sub-interface, send $\langle \texttt{m:done:p}{:}\varphi \rangle$ to $\mathcal{F}_{\text{ABB}}$.

Upon receiving $\langle \texttt{deliver} \rangle$ from $\mathcal{A}$ via the mth sub-interface, send $\langle \texttt{m:done:q}{:}\varphi \rangle$ to $\mathcal{F}_{\text{ABB}}$.

$\mathcal{P}$ corrupted first. Without loss of generality, we may assume that whenever $\mathcal{P}$ gets corrupted, all of her subroutines are immediately corrupted as well. We only need to show how $\mathcal{S}$ operates in the case that $\mathcal{P}$ starts out corrupted: if $\mathcal{P}$ gets corrupted later, $\mathcal{S}$ starts by recovering $\mathcal{P}$'s input

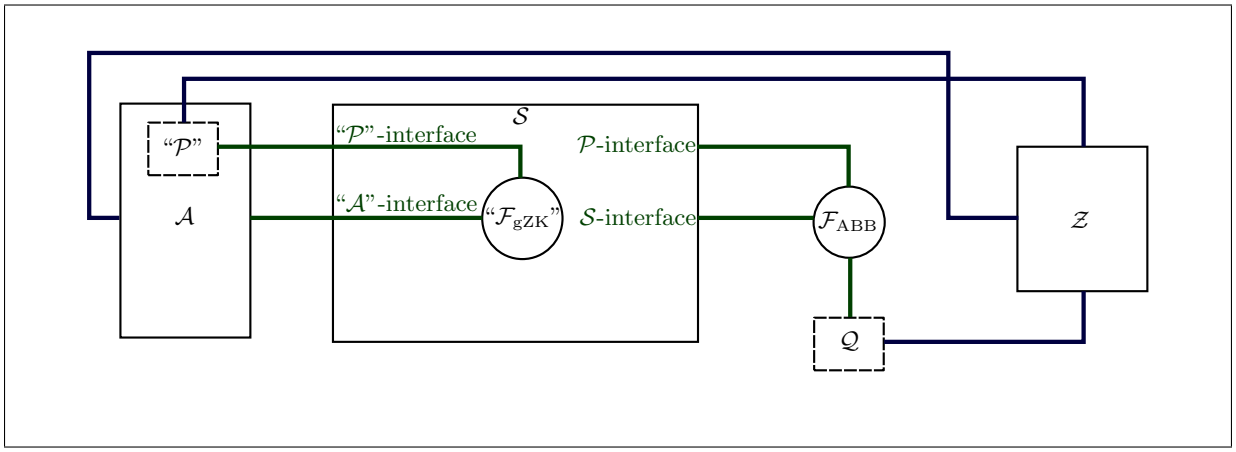

Fig. 9: Construction of $\mathcal{S}$ in case $\mathcal{P}$ is corrupted.

by sending one $\langle$`ip:expose:`$\varphi\rangle$ message to $\mathcal{F}_{\text{ABB}}$ for each *Input by* $\mathcal{P}$ instruction that has already processed the $\langle$`ip:send:`$\varphi\rangle$ message. The simulator $\mathcal{S}$ also recovers $\mathcal{P}$'s external witnesses from each *Proof by* $\mathcal{P}$ instruction that has processed the $\langle$`pp:send:`$\varphi\rangle$ message but not the $\langle$`pp:lock:`$\varphi\rangle$ message by sending $\langle$`pp:expose:`$\varphi\rangle$ to $\mathcal{F}_{\text{ABB}}$. Now, $\mathcal{S}$ internally restarts the simulation of "$\mathcal{P}$" from the beginning until the point where she was corrupted. For all instructions except the *Proof by* $\mathcal{P}$ that have already processed the $\langle$`pp:lock:`$\varphi\rangle$ message, $\mathcal{S}$ can perfectly re-create "$\mathcal{P}$"'s input. For the *Proof by* $\mathcal{P}$ instructions that have processed the $\langle$`pp:lock:`$\varphi\rangle$ message, $\mathcal{S}$ can use arbitrary input (of the correct length!) for "$\mathcal{P}$", since that input will be erased by "$\mathcal{P}$" and since it does not affect the remainder of the protocol. Finally, $\mathcal{S}$ hands over the internal state of "$\mathcal{P}$" to $\mathcal{A}$. This state is perfectly consistent with $\mathcal{A}$'s view so far. Of course, we will have to deal with the possibility that $\mathcal{Q}$ is corrupted later on, which we tackle later in this section.

Overview. Recall that when $\mathcal{P}$ is corrupted, $\mathcal{S}$ must play $\mathcal{P}$ for $\mathcal{F}_{\text{ABB}}$ based on the actions of "$\mathcal{P}$" (assumed by $\mathcal{A}$), and must play "$\mathcal{Q}$" for $\mathcal{A}$ *without knowing the correct input of* $\mathcal{Q}$. See Figure 9.

In constructing the simulator, we maintain the invariant that $\mathcal{S}$ knows the value of "$\mathcal{P}$"'s shares $SP[k]$ when these are ready to be used, i.e., $k \in RP$.

Input from $\mathcal{P}$. For this instruction, $\mathcal{S}$ simply needs to extract the input of "$\mathcal{P}$" from the messages flowing on the "$\mathcal{P}$"-interface or the "$\mathcal{A}$"-interface. We show here the exact behaviour of $\mathcal{S}$ for completeness.

Upon receiving $\langle$`send`$, \langle CP[k]\rangle, \langle v\rangle, \ldots\rangle$ through the "$\mathcal{P}$"-interface, save v as $SP[k]$. Send $\langle$`send`$\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`reset`$, \langle CP[k]\rangle, \langle v\rangle, \ldots\rangle$ through the "$\mathcal{A}$"-interface, update $SP[k]$ with the new value of v. Send $\langle\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`expose`$\rangle$ through the "$\mathcal{A}$"-interface, send $\langle$`expose`$, \langle CP[k]\rangle, \langle SP[k]\rangle\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`ip:ready:`$\varphi\rangle$ through the $\mathcal{S}$-interface, send $\langle$`ready`$\rangle$ through the "$\mathcal{Q}$"-interface.

Upon receiving $\langle \texttt{lock} \rangle$ through the "$\mathcal{A}$"-interface, send $\langle \texttt{ip:send:}\varphi, SP[k] \rangle$ through the $\mathcal{P}$-interface. Wait for $\langle \texttt{ip:send:}\varphi \rangle$ through the $\mathcal{S}$-interface, send $\langle \texttt{ip:lock:}\varphi \rangle$ through the $\mathcal{S}$-interface. Wait for $\langle\rangle$ through the $\mathcal{S}$-interface, send $\langle\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle \texttt{deliver}, \ell \rangle$ through the "$\mathcal{A}$"-interface, send $\langle \texttt{ip:deliver:}\varphi \rangle$ through the $\mathcal{S}$-interface.

Upon receiving $\langle \texttt{done} \rangle$ through the "$\mathcal{A}$"-interface, send $\langle \texttt{ip:done:}\varphi \rangle$ through the $\mathcal{S}$-interface. Wait for $\langle \texttt{ip:done:}\varphi \rangle$ through the $\mathcal{P}$-interface, send $\langle \texttt{done} \rangle$ through the "$\mathcal{P}$"-interface.

Input from $\mathcal{Q}$. For this instruction, $\mathcal{S}$ needs to generate an equivocable commitment to $\mathcal{Q}$'s input, which $\mathcal{S}$ doesn't know.

The construction of $\mathcal{S}$ in response to the `send`, `ready`, `lock` and `done` messages is straightforward.

Upon receiving $\langle \texttt{deliver}, \ell \rangle$ through the "$\mathcal{A}$"-interface, commit to 0 using an equivocable commitment: $(SQ[k], XQ[k]) \xleftarrow{\$} \mathsf{Com}(0)$, and send $\langle \texttt{deliver}, \langle SQ[k] \rangle \rangle$ through the "$\mathcal{P}$"-interface.

Output to $\mathcal{P}$. In this instruction, $\mathcal{S}$ recovers the value that is output from the circuit from $\mathcal{F}_{\text{ABB}}$ just in time to be able to play "$\mathcal{Q}$" in a consistent manner.

The construction of $\mathcal{S}$ in response to the `q`, `ready`, and `done` messages is straightforward.

Upon receiving $\langle \texttt{lock} \rangle$ through the "$\mathcal{A}$"-interface, $\mathcal{S}$ sends $\langle \texttt{op:lock:}\varphi \rangle$ through the $\mathcal{S}$-interface and expects $\langle\rangle$ through the $\mathcal{S}$-interface. Then, $\mathcal{S}$ sends $\langle \texttt{op:deliver:}\varphi \rangle$ through the $\mathcal{S}$-interface, and expects $\langle \texttt{op:deliver:}\varphi, V[k] \rangle$ through the $\mathcal{P}$-interface, thereby recovering $V[k]$.

Upon receiving $\langle \texttt{deliver}, \ell \rangle$ through the "$\mathcal{A}$"-interface, $\mathcal{S}$ sends $\langle \texttt{deliver}, V[k] - SP[k] \rangle$ through the "$\mathcal{P}$" interface.

Exponentiated output to $\mathcal{P}$. For this instruction, $\mathcal{S}$ behaves similarily than for the *Output to $\mathcal{P}$* instruction. The difference is that it receives $\langle \texttt{ep:deliver:}\varphi, \mathtt{g}^{V[k]} \rangle$ through the $\mathcal{P}$-interface, and sends $\langle \texttt{deliver}, \mathtt{g}^{V[k]}/\mathtt{g}^{SP[k]} \rangle$ through the "$\mathcal{P}$"-interface.

Output to $\mathcal{Q}$. The simulation of this instruction is straightforward.

Exponentiated output to $\mathcal{Q}$. The simulation of this instruction is straightforward.

Linear combination. The simulation of this instruction is straightforward. Additionally, $\mathcal{S}$ computes "$\mathcal{P}$"'s share $SP[k_0]$ based on the values of $SP[k_i]$ (which $\mathcal{S}$ knows).

Proof by $\mathcal{P}$. The simulation of this instruction is also relatively straightforward. Futhermore, we explain why $\mathcal{S}$ also works properly in the security proof of Π_{gABB}, the realization of $\mathcal{F}_{\text{gABB}}$ which allows one to prove the *existence* of external witnesses.

$\mathcal{S}$'s reaction to the `ready`, `done`, and `deliver` messages is straightforward.

Upon receiving $\langle \texttt{send}, x, w_k, \ldots \rangle$ through the "$\mathcal{P}$"-interface, save x and w_k (for Π_{gABB}, $\mathcal{S}$ does not save w_e). Send $\langle \texttt{send}, \ell(x, w_k) \rangle$ through the "$\mathcal{A}$" interface.

Upon receiving $\langle \texttt{reset}, x, w_k, \ldots \rangle$ through the "$\mathcal{A}$"-interface, save the updated x and w_k (again, for Π_{gABB}, $\mathcal{S}$ does not save w_e). Send $\langle\rangle$ through the "$\mathcal{A}$" interface.

Upon receiving $\langle \texttt{expose} \rangle$ through the "$\mathcal{A}$"-interface, send $\langle \texttt{expose}, x, w_k \rangle$ through the "$\mathcal{A}$"-interface (for Π_{gABB}: recall that w_k is not sent to $\mathcal{A}$).

Upon receiving $\langle \texttt{lock} \rangle$ through the "$\mathcal{A}$" interface, send $\langle \texttt{pp:send:}\varphi, x, |w_k|, w_k, \mathtt{0}, \langle\rangle \rangle$ through the $\mathcal{P}$-interface. (For Π_{gABB}, no modifications are necessary: recall that $\mathcal{F}_{\text{gABB}}$ does not check

if the predicate is satisfied; sending a wrong w_e to $\mathcal{F}_{\text{gABB}}$ is indistinguishable from sending the correct w_e.) Expect $\langle \texttt{pp:send:}\varphi, \ell \rangle$ through the $\mathcal{S}$-interface. Send $\langle \texttt{pp:lock:}\varphi \rangle$ through the $\mathcal{S}$-interface, expect $\langle\rangle$ through the $\mathcal{S}$-interface. Send $\langle\rangle$ through the "$\mathcal{A}$"-interface.

Proof by $\mathcal{Q}$. The simulation of this instruction is straightforward.

Multiplication. See the next paragraph for the behaviour of $\mathcal{S}$ inside Π_{mul}: $\mathcal{S}$ recovers "$\mathcal{P}$"'s private outputs $\tilde{u}$ and u. The simulation of the remainder of this instruction is straightforward. Additionally, $\mathcal{S}$ computes "$\mathcal{P}$"'s share $SP[k_0]$ based on the values of $SP[k_1]$, $SP[k_2]$, $\tilde{u}$ and u (which $\mathcal{S}$ knows).

Π_{mul}. Recall that $\mathcal{S}$ knows "$\mathcal{P}$"'s input a from the *Multiplication* instruction. The construction of the simulator is straightforward, expect for three changes where $\mathcal{S}$ deviates from the honest execution.

First, when receiving $\langle \texttt{send}, \langle \mathsf{E}_w, \mathsf{pk} \rangle, w, \ldots \rangle$ through the $\langle \texttt{cm1}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ saves the value w instead of discarding it.

Second, instead of sending the correct $\langle \texttt{deliver}, \langle \mathfrak{C}_s, \mathfrak{C}_t, \mathsf{E}_y \rangle \rangle$ message through the $\langle \texttt{cm2}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ chooses a random y, encrypts it $(\mathsf{E}_y, \mathsf{r}_y) \xleftarrow{\$} \mathsf{Enc}(y)$, and delivers the inconsistent E_y. Note that $\mathcal{S}$ will never have to show r_y, since "$\mathcal{Q}$" would have erased that value already.

Third, instead of sending the correct $\langle \texttt{deliver}, \delta \rangle$ message through the $\langle \texttt{cm4}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ chooses a random δ, and delivers it.

Finally, $\mathcal{S}$ recovers "$\mathcal{P}$"'s output as follows: $u \leftarrow \delta \cdot a + y$ (with the values of y and δ that $\mathcal{S}$ chose). Notice that $\mathcal{S}$ did not use "$\mathcal{Q}$"'s input b.

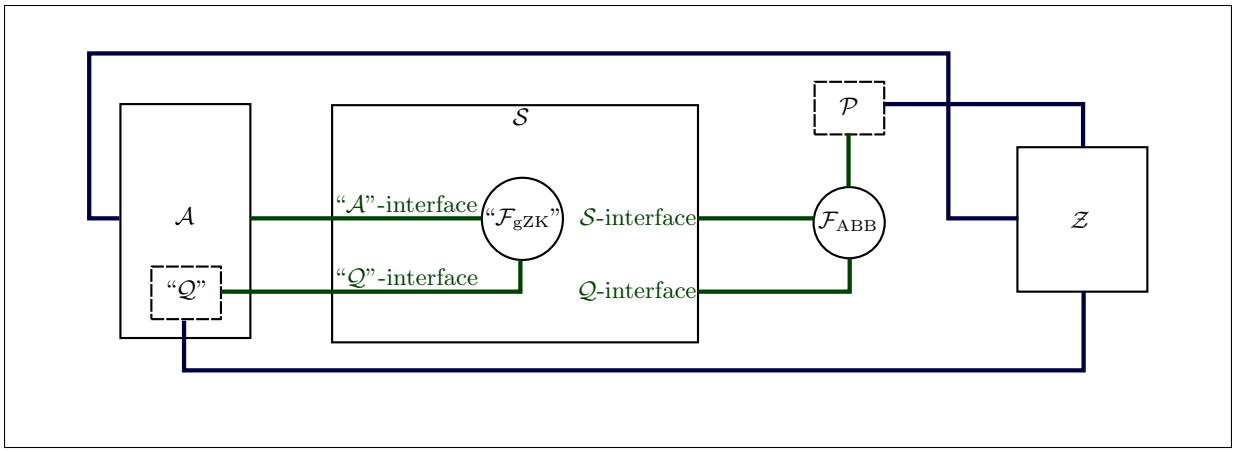

Fig. 10: Construction of $\mathcal{S}$ in case $\mathcal{Q}$ is corrupted.

$\mathcal{Q}$ corrupted first. This case is similar to the case where $\mathcal{P}$ was corrupted first. For all instructions except Π_{mul}, $\mathcal{S}$'s behaviour can be inferred from its behaviour in the case where $\mathcal{P}$ was corrupted first. See Figure 10.

Π_{mul}. Recall that $\mathcal{S}$ knows "$\mathcal{Q}$"'s input b from the *Multiplication* instruction. The construction of the simulator is straightforward, expect for two changes where $\mathcal{S}$ deviates from the honest execution.

First, when receiving $\langle \texttt{send}, \langle \mathfrak{C}_s, \mathfrak{C}_t, \mathsf{E}_y \rangle, s, \ldots \rangle$ through the $\langle \texttt{cm2}, \lambda \rangle$ sub-interface of the "$\mathcal{Q}$"-interface, $\mathcal{S}$ saves the value s instead of discarding it.

Second, instead of sending the correct $\langle \texttt{deliver}, \langle \mathfrak{C}_y, \sigma \rangle \rangle$ message through the $\langle \texttt{cm3}, \lambda \rangle$ sub-interface of the "$\mathcal{Q}$"-interface, $\mathcal{S}$ chooses a random σ, and delivers that. $\mathcal{S}$ will never have to show sk, since "$\mathcal{P}$" would have erased that value already.

Finally $\mathcal{S}$ recovers "$\mathcal{Q}$"'s output v as follows: $s \leftarrow b - \delta; t \leftarrow y - w \cdot s; v \leftarrow \sigma \cdot s - t$ (using the value of σ that $\mathcal{S}$ chose). Notice that $\mathcal{S}$ did not use "$\mathcal{P}$"'s input a.

Adjusting "$\mathcal{Q}$"'s state when $\mathcal{Q}$ is corrupted second. When $\mathcal{Q}$ gets corrupted second, $\mathcal{S}$ needs to come up with a believable internal state for "$\mathcal{Q}$". In order to do so, $\mathcal{S}$ sets "$\mathcal{Q}$"'s internal state (SQ, XQ and local variables) in each instruction, in the order in which they were processed, as follows:

Input from $\mathcal{P}$. The adjustements to make are straightforward.

Input from $\mathcal{Q}$. If $\mathcal{S}$ accepted the $\langle \texttt{iq:send}{:}\varphi \rangle$ message, $\mathcal{S}$ recovers the orignal input of $\mathcal{Q}$: $\mathcal{S}$ sends $\langle \texttt{iq:expose}{:}\varphi \rangle$ through the $\mathcal{S}$-interface, and expects $\langle \texttt{iq:expose}{:}\varphi, SQ[k] \rangle$ through the $\mathcal{S}$-interface. $\mathcal{S}$ now *adjusts the opening $XQ[k]$ of the commitment $CQ[k]$*, i.e., $\mathcal{S}$ uses the $\mathsf{Trapdoor}$ to find a new value of the opening $XQ[k]$ of the commitment $CQ[k]$ so that $\mathsf{ComVfy}(CQ[k], XQ[k], SQ[k]) = \text{true}$.

Output to $\mathcal{P}$. The adjustements to make are straightforward. Notice that the value $(V[k] - SP[k])$ that $\mathcal{S}$ delivered is equal to $SQ[k]$ as expected, unless $\mathcal{A}$ somehow managed to equivocate one of her commitments.

Output to $\mathcal{Q}$. The adjustements to make are straightforward.

Exponentiated output to $\mathcal{P}$. The adjustements to make are straightforward. Notice that the value $(\mathsf{g}^{V[k]}/\mathsf{g}^{SP[k]})$ that $\mathcal{S}$ delivered is equal to $\mathsf{g}^{SQ[k]}$ as expected, unless $\mathcal{A}$ somehow managed to equivocate one of her commitments.

Exponentiated output to $\mathcal{Q}$. The adjustements to make are straightforward.

Proof by $\mathcal{P}$. The adjustements to make are straightforward.

Proof by $\mathcal{Q}$. If $\mathcal{S}$ accepted the $\langle \texttt{pq:send}{:}\varphi \rangle$ message, but did not yet deliver the $\langle \texttt{pq:lock}{:}\varphi \rangle$ message, $\mathcal{A}$ still has a chance to send $\langle \texttt{expose} \rangle$ to $\mathcal{S}$ through the "$\mathcal{A}$"-interface, and therefore $\mathcal{S}$ needs to recover $\mathcal{Q}$'s input, i.e., x and w_k. $\mathcal{S}$ sends $\langle \texttt{pq:expose}{:}\varphi \rangle$ through the $\mathcal{S}$-interface, and expects $\langle \texttt{pq:expose}{:}\varphi, x, w_k \rangle$. $\mathcal{S}$ saves x and w_e.

The remainder of the adjustements to make are straightforward.

Linear combination. $\mathcal{S}$ computes "$\mathcal{Q}$"'s share $SQ[k_0]$ based on the other shares $SQ[k_i]$, and adjusts the opening $XQ[k_0]$.

Multiplication. $\mathcal{S}$ performs the necessary adjustements inside the Π_{mul} subroutine. $\mathcal{S}$ will recover "$\mathcal{Q}$"'s output $\tilde{v}$ and v from Π_{mul}.

$\mathcal{S}$ computes "$\mathcal{Q}$"'s share $SQ[k_0]$ based on $SQ[k_1]$, $SQ[k_2]$, $\tilde{v}$, and v. $\mathcal{S}$ adjusts the opening $XQ[k_0]$.

The remainder of the adjustements to make are straightforward.

Π_{mul}. If $\mathcal{Q}$ gets corrupted before the delivery of the $\langle \texttt{deliver}, \langle \mathfrak{C}_s, \mathfrak{C}_t, \mathsf{E}_y \rangle \rangle$ message through the $\langle \texttt{cm2}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, the adjustements to make are trivial.

If $\mathcal{Q}$ gets corrupted after the delivery of the $\langle \texttt{deliver}, \langle \mathfrak{C}_s, \mathfrak{C}_t, \mathsf{E}_y \rangle \rangle$ message through the $\langle \texttt{cm2}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, but before the delivery of the $\langle \texttt{deliver}, \delta \rangle$ message through the $\langle \texttt{cm4}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ is committed to its "incorrect" E_y. Since $\mathcal{A}$ can decrypt that value, $\mathcal{S}$ is therefore also committed to y. $\mathcal{S}$ needs to find s and t consistent with y: $\mathcal{S}$ sets s at random, and computes $t \leftarrow y - w \cdot s$. $\mathcal{S}$ then adjusts the opening $\mathfrak{r}_s$ and $\mathfrak{r}_t$ using the trapdoor. $\mathcal{S}$ does not need to compute r_t, as it can claim that "$\mathcal{Q}$" securely erased that value already. E_t can be re-computed from E_y, E_w and s.

If $\mathcal{Q}$ gets corrupted after the delivery of the $\langle \texttt{deliver}, \delta \rangle$ message through the $\langle \texttt{cm4}, \lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ is committed to E_y (i.e., to y) and to δ. $\mathcal{S}$ sets $s \leftarrow b - \delta$, $t \leftarrow y - w \cdot s$, $v \leftarrow \sigma \cdot s - t$, and adjusts the openings $\mathfrak{r}_t$, $\mathfrak{r}_s$, and $\mathfrak{r}_v$ using the trapdoor. $\mathcal{S}$ now knows the correct output of "$\mathcal{Q}$".

The remainder of the adjustements to make are straightforward.

Adjusting "$\mathcal{P}$"'s state when $\mathcal{P}$ is corrupted second. This case is similar to the case where $\mathcal{Q}$ was corrupted second. For all instructions except Π_{mul}, $\mathcal{S}$'s behaviour can be inferred from $\mathcal{S}$'s behaviour in the case where $\mathcal{Q}$ was corrupted second.

Π_{mul}. If $\mathcal{P}$ gets corrupted before the delivery of the $\langle \texttt{deliver}, \langle \mathfrak{C}_y, \sigma \rangle \rangle$ message through the $\langle \texttt{cm3}, \lambda \rangle$ sub-interface of the "$\mathcal{Q}$"-interface, the adjustements to make are trivial.

If $\mathcal{P}$ gets corrupted after the delivery of that message, $\mathcal{S}$ is bound to w (via E_w and pk) and to σ (which was delivered to $\mathcal{A}$). However at this point, $\mathcal{S}$ can claim that "$\mathcal{P}$" already securely erased r_w and sk, and so it can get away with revealing a value of w that is inconsistent with E_w and pk (the semantic security of the encryption hides that inconsistency, as proven more formally later). $\mathcal{S}$ now needs to adjust the values y and w in "$\mathcal{P}$"'s internal state: $\mathcal{S}$ computes $w' \leftarrow a - \sigma$ and $y' \leftarrow (w' - w) \cdot s + y$, and replaces y by y' and w by w' in "$\mathcal{P}$"'s internal state. Furthermore $\mathcal{S}$ adjusts $\mathfrak{r}_y$ using the trapdoor. Finally, $\mathcal{S}$ recomputes "$\mathcal{P}$"'s output u: $u \leftarrow \delta \cdot a + y'$ and adjusts the opening $\mathfrak{r}_u$ using the trapdoor.

Both parties corrupted. Once $\mathcal{S}$ has handed over the complete *non-erased* internal state of the second corrupted party to $\mathcal{A}$, the simulation is trivial: $\mathcal{S}$ runs "$\mathcal{F}_{\text{gZK}}$" and "$\mathcal{F}_{\text{ach}}$" honestly, and does not send any messages to $\mathcal{F}_{\text{ABB}}$. See Figure 11.

Proof of indistinguishability. We are going to define a sequence of games Game_1 to $\mathsf{Game}_{N_{\text{games}}}$, as described by Shoup [Sho04]. In the first game, everything is distributed as in the protocol Π_{ABB}, whereas in the last game everything is distributed as in the ideal world $\mathcal{F}_{\text{ABB}}$. By the piling-up lemma, the advantage of $\mathcal{Z}$ is less than the sum of the advantages in distinguishing between Game_i and Game_{i+1}. We are going to prove that $\mathcal{Z}$ only has negligible

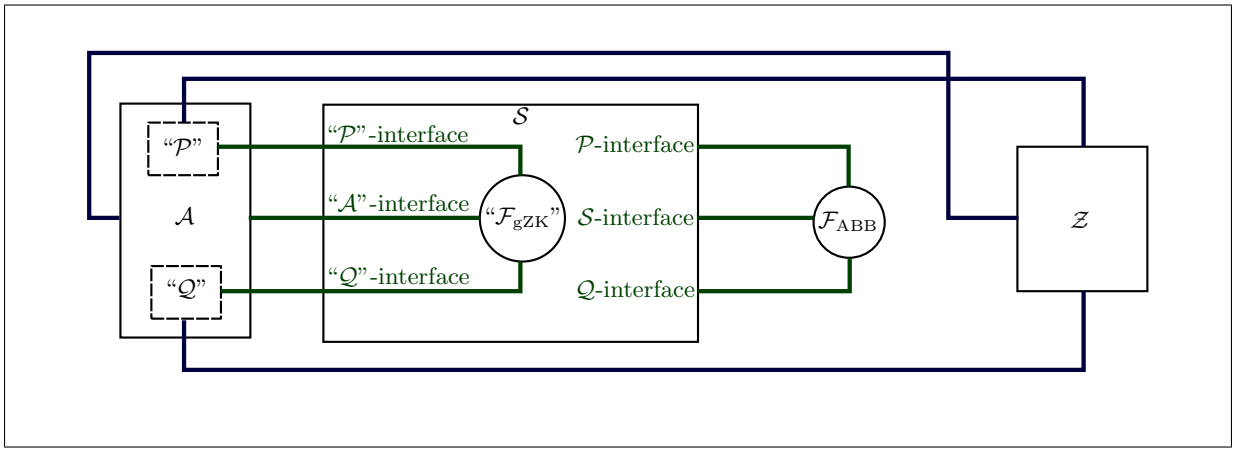

Fig. 11: Construction of $\mathcal{S}$ in case all parties are corrupted.

advantage in distinguishing between two consecutive games, based either on a reduction to a hard cryptographic problem, or by "failure events" happening with negligible probability. As long as the number of games is polynomial w.r.t. the security parameter, the total advantage of $\mathcal{Z}$ is negligible.

We must stress that in all intermediate games, the simulator $\mathcal{S}_i$ receives the inputs of all honest parties (i.e., we are not in the "ideal world" yet). We only require that the simulator of the last game, which is equivalent to the "ideal" world, does not make use of these inputs.

Game_1*:* As observed in the previous paragraph, $\mathcal{S}_1$ receives the input of all honest parties. $\mathcal{S}_1$ simply runs the parties it controls honestly, and exposes their internal state to $\mathcal{A}$ when corrupted. $\mathcal{S}_1$ generates the CRS honestly using ComGen_1. By construction, this setting is perfectly indistinguishable from the $(\mathcal{F}_{\text{gZK}}, \mathcal{F}_{\text{ach}})$-hybrid "real" world Π_{ABB}.

Game_2*:* $\mathcal{S}_2$ runs like $\mathcal{S}_1$, except that it aborts if $\mathcal{A}$'s output is inconsistent with its inputs at any time during the protocol. The probability that $\mathcal{S}_2$ aborts is at most the probability that the commitment was not binding after all, which is negligible.

Game_3*:* $\mathcal{S}_3$ runs like $\mathcal{S}_2$, except that now it chooses the CRS with ComGen'_0 instead of ComGen_1. The commitment scheme is now perfectly hiding, and $\mathcal{S}_3$ can now efficiently equivocate commitments using the trapdoor information. The advantage that $\mathcal{Z}$ has in distinguishing between Game_3 and Game_2 is equal to its advantage in breaking the semantic security game of Elgamal encryption, which in turn is equal to its advantage in breaking DDH in the group modulo $\mathfrak{p}$ generated by $\mathfrak{g}$, which is negligible.

Game_4*:* $\mathcal{S}_4$ runs as $\mathcal{S}_3$, except that during the Π_{mul} subroutines, it behaves as described earlier in this section, i.e., it ignores the input of the parties it controls during the Π_{mul} protocol and reconstructs a plausible history upon corruption. It is easy to see that the only way $\mathcal{Z}$ can get an advantage in distinguishing between Game_4 and Game_3 is, if upon corruption of "$\mathcal{P}$", it notices that the values w, E_w and pk are inconsistent. We now argue that the advantage of $\mathcal{Z}$ is at most the advantage $\mathcal{Z}$ has in breaking the semantic security of the Camenisch-Shoup encryption times the number of Π_{mul} sub-protocols that "$\mathcal{P}$" started, which is still negligible.

Let $N_{\Pi_{\text{mul}}}$ be the number of times "$\mathcal{P}$" calls Π_{mul} in Π_{ABB}.

We are going to define a polynomial number of hybrid games $\mathsf{Game}_{3:0}$ to $\mathsf{Game}_{3:N_{\Pi_{\text{mul}}}}$, where in $\mathsf{Game}_{3:i}$ $\mathcal{S}$ behaves like $\mathcal{S}_4$ for the first i calls to Π_{mul}, and like $\mathcal{S}_3$ for the subsequent calls. Clearly $\mathsf{Game}_{3:0}$ is exactly Game_3 and $\mathsf{Game}_{3:N_{\Pi_{\text{mul}}}}$ is exactly Game_4.

If there exists $\mathcal{Z}$ which has non-negligible advantage γ in distinguishing between Game_4 and Game_3, then there must exist another environment $\mathcal{Z}'$ and a value $i \in \mathbb{N}^*_{N_{\Pi_{\text{mul}}}}$ such that $\mathcal{Z}'$ has advantage $\gamma/N_{\Pi_{\text{mul}}}$ in distinguishing between $\mathsf{Game}_{3:i}$ and $\mathsf{Game}_{3:i-1}$, which is still a non-negligible advantage.

We now show how $\mathcal{S}$ can use such an $\mathcal{Z}'$ to break the semantic security of the encryption function with advantage $\gamma/N_{\Pi_{\text{mul}}}$.

In the ith run of the Π_{mul} protocol on behalf of "$\mathcal{P}$", instead of computing E_w honestly, $\mathcal{S}$ submits two plaintexts w and w' to the challenger of the semantic security game, yielding a challenge plaintext $\mathsf{E}_{\bar{w}}$ which is either equal to the consistent E_w or the inconsistent $\mathsf{E}_{w'}$, and a public key pk. Recall that $\mathcal{S}$ does not need to know the value of w in order to properly run the simulation; w is only needed upon corruption of $\mathcal{P}$. $\mathcal{S}$ now uses $\mathsf{E}_{\bar{w}}$ instead of E_w.

If $\mathcal{P}$ becomes corrupted before the delivery of σ in the ith Π_{mul} protocol, then $\mathcal{S}$ aborts the simulation (it cannot produce a convincing value of sk) and returns a random guess to the challenger. Since in this run the view of $\mathcal{Z}'$ would have been perfectly indistinguishable, $\mathcal{S}$ does not lose any advantage by aborting.

If however $\mathcal{P}$ become corrupted after the delivery of σ (or not at all), then $\mathcal{S}$ will produce an internal state for "$\mathcal{P}$" that contains, among others: w, $\mathsf{E}_{\bar{w}}$, and pk. $\mathcal{S}$ returns the same guess as $\mathcal{Z}'$ to the challenger: if $\mathsf{E}_{\bar{w}} = \mathsf{E}_w$ then the view of $\mathcal{Z}'$ is exactly that of $\mathsf{Game}_{3:i-1}$, and if $\mathsf{E}_{\bar{w}} = \mathsf{E}_{w'}$ then the view is exactly that of $\mathsf{Game}_{3:i}$.

The distinguishing advantage of $\mathcal{Z}$ between Game_4 and Game_3 is therefore negligible.

Game_5*:* $\mathcal{S}_5$ runs as described earlier in this section for all instructions, and not just Π_{mul}. By construction, $\mathcal{S}_5$ does not need to know the input of the honest parties ($\mathcal{S}_5$ extracts it from $\mathcal{F}_{\text{ABB}}$) and $\mathcal{S}_5$'s behaviour is perfectly indistinguishable from $\mathcal{S}$'s behaviour in the ideal world. The difference between Game_5 and Game_4 is zero, thanks to the perfectly hiding commitments.

Conclusion. This concludes the proof of Lemma 1, i.e., Π_{ABB} securely implements $\mathcal{F}_{\text{ABB}}$ for all *nice* environments.

7 Related Work and Comparison

There is an extensive literature on the subject of multi-party computation (MPC); however, most of these settings consider only the case of an honest majority, which is not helpful for the two-party case.

Canetti et al. [CLOS02] present the first MPC protocols for general functionalities that are secure with dishonest majority in the UC framework; however, these protocols are rather a proof of concept, i.e., they are not at all practical, as they rely on generic zero-knowledge proofs.

More efficient MPC protocols for evaluating *boolean* circuits, secure with dishonest majority, have been designed [LP07,LPS08,NO09,PSSW09]. Impressive results have been obtained in particular for the evaluation of the AES block cipher [PSSW09,DK10,DKL+12,KSS12,NNOB12]. While such protocols could be used to evaluate arithmetic circuits modulo n, a heavy price would

have to be paid: each gate in the arithmetic circuit would "blow up" into many boolean gates, resulting in an impractical protocol.

The first practical protocols for evaluating arithmetic circuits modulo n were presented by Cramer et al. [CDN01] (CDN-protocol) and Damgård and Nielsen [DN03] (DN-protocol). While both protocols assume an honest majority, they can be shown to be secure in the two-party case (as noted by Ishai et al. [IPS09,IPS08]) if one relaxes the requirement for fair delivery of messages (fair delivery is impossible in the two-party case). Both protocols have stronger set-up assumptions than ours: they assume the existence of a trusted third party that distributes shares of the secret key to all parties. The CDN-protocol is only *statically* secure and is not UC-secure, and we therefore exclude it from our comparison. The DN-protocol is *adaptively* secure (with erasures) in the UC model (secure *without erasures* only in the honest majority case), and is slightly (about 30%) slower than ours.

Ishai et al. [IPS08,IPS09] present protocols for evaluating arithmetic circuits in several algebraic rings, including one for the ring $\mathbb{Z}_{\mathsf{n}}$ for a composite n. These protocols achieve security with a dishonest majority, and are secure with respect to *adaptive* corruptions (assuming erasures), but only against *honest-but-curious* adversaries. They note that standard techniques can be used to make their protocols secure also for *malicious* adversaries, however it is not clear if the resulting construction will be practical. Our protocol draws on ideas from their construction, however we are able to achieve a significant speed-up compared to a naive implementation using "standard techniques" by ensuring that all commitments live in $\mathbb{Z}_{\mathsf{n}}$ and by using the short-key variant of the homomorphic encryption scheme.

Damgård and Orlandi [DO10a] (DO-protocol), as well as Bendlin et al. [BDOZ11] (BDOZ-protocol), give protocols for evaluating arithmetic circuits modulo a prime p. Damgård et al. [DPSZ12] (SPDZ-protocol) later improved upon the BDOZ-protocol. These protocols divide the workload into a computationally intensive *pre-processing* phase and a much lighter *on-line* phase. The pre-processing phase is statically secure, however the on-line phase can be made adaptively secure (in the UC-model) [DO10a,BDOZ11,DPSZ12]. These papers optimize the runtime of the on-line phase (the BDOZ- and SPDZ-protocols make use of local additions and multiplications only). In the pre-processing phase of these protocols, it is necessary to prepare for many multiplications gates (about 80 in the BDOZ-, several hundred in the DO-, and tens of thousands in the SPDZ-protocol) making these protocols impractical for small circuits. This pre-processing phase takes several minutes even for reasonable security parameters. Our protocol is better suited for small circuits.

Even for large circuits, the computational complexity of our protocol is about 3.3 times lower than that of the BDOZ- and DO-protocols. It must be noted that the BDOZ- and DO-protocols have slightly weaker setup assumptions than ours: they only require a *random string* as the CRS, while we also need an *RSA modulus with unknown factorization* as a system parameter. (This is not a huge drawback of our protocol, see Section 3.1.)

The SPDZ-protocol is about an order of magnitude faster than our protocol, however, unlike the BDOZ-, DO-, and our protocols, it cannot evaluate reactive circuits, severely limiting its applicability in the real world. It also requires a trusted key setup, which is a stronger setup assumption than ours. (Concurrently to our work, Damgård et al. [DKL$^+$13] lifted the restriction on reactive circuits, but only in the *random oracle model*. They also lifted the restriction on the trusted key setup but only for *covert* security.)

	Amortized runtime per multiplication gate	with $\mathsf{s}=80$
This work	$(90 \cdot \mathsf{s} + 200 \cdot \mathrm{lb}\,\mathsf{n})\ \mathrm{exp}.\mathsf{n} + (\ 66 \cdot \mathsf{s} + \ 40.5 \cdot \mathrm{lb}\,\mathsf{n})\ \mathrm{exp}.\mathsf{n}^2$	**602 ms**
2-party DN-protocol [DN03]	$(216 \cdot \mathsf{s} + 130 \ \cdot \mathrm{lb}\,\mathsf{n})\ \mathrm{exp}.\mathsf{n}^2$	862 ms
DO-protocol [DO10a]	$(2004 \cdot \mathsf{s} + 151 \cdot \mathsf{s}^2)\ \mathrm{exp}.\mathsf{n} + (\ 84 \cdot \mathsf{s} + \ 88 \ \cdot \mathrm{lb}\,\mathsf{n})\ \mathrm{exp}.\mathsf{n}^2$	2025 ms
BDOZ-protocol [BDOZ11]	$(256 \cdot \mathsf{s} + 368 \ \cdot \mathrm{lb}\,\mathsf{n})\ \mathrm{exp}.\mathsf{n}^2$	2303 ms

Table 1. Estimated amortized runtime per multiplication in various protocols. The numbers in the last column are for $\mathsf{s} = 80$, $\mathrm{lb}\,\mathsf{n} = 1248$, $\mathrm{exp}.\mathsf{n} = 1.3\,\mu\mathrm{s}$, and $\mathrm{exp}.\mathsf{n}^2 = 4.8\,\mu\mathrm{s}$. Results for our work use the *optimized* variant of our *Multiplication* instruction. Results for the DO-protocol and the BDOZ-protocol are for circuits having a multiple of $4.8 \cdot \mathsf{s}$ and s multiplication gates, respectively; the performance of these protocols degrades dramatically for smaller circuits. For the DO-protocol we used parameters $\lambda = 0.25$ and $B = 3.6 \cdot \mathsf{s}$.

None of the UC-secure protocols discussed have an equivalent to the *Proof* instruction in their ideal functionality. This makes it hard to compose them with other protocols because of the issue with non-committed inputs in a 2-party setting, as dicussed in the introduction, thus negating some of the advantages of working in the UC model.

7.1 Efficiency Comparison

Table 1 summarizes the amortized runtimes per multiplication gate of our protocol, the DN- (when run as a 2-party protocol), the DO-, and the BDOZ-protocols. We assume that the runtime of an exponentiation with a fixed modulus length scales linearly with the size of the exponent. Let $\mathrm{exp}.\mathsf{n}$ denote the runtime per bit in the exponent of an exponentiation modulo n or modulo $\mathfrak{p}$,[7] and similarily $\mathrm{exp}.\mathsf{n}^2$ for exponentiations modulo n^2. Let $\mathrm{lb}\,\mathsf{n}$ be equal to $\log_2(\mathsf{n})$. Let s be the security parameter. For each protocol, we counted the number of exponentiations with an exponent of at least s bits. Faster operations, in particular multiplications and divisions, are ignored. We also ignored the time needed for secure channel setup, did not consider multi-exponentiations, and ignored network delay. We provide an estimate of the runtime when run with the "smallest general purpose" security level of the Ecrypt-II recommendations [BCC+11] ($\mathsf{s} = 80$, $\mathrm{lb}\,\mathsf{n} = 1248$) on a standard laptop with a 64-bit operating system.[8]

For a fair comparison, we replace all Paillier encryptions [Pai99] in the protocols we compare with by Paillier encryptions *with short randomness*. The encryption function is thus changed as follows: $\mathsf{r} \xleftarrow{\$} \mathbb{Z}_{\lfloor\sqrt{\mathsf{n}}\rfloor}$, $c \leftarrow (1+\mathsf{n})^m \mathsf{g}^{\mathsf{r}} \pmod{\mathsf{n}^2}$; output c. (Where $\mathsf{g} = (\mathsf{g}')^{\mathsf{n}}$ is pre-computed and part of the public key.)

7.2 Comments about the Efficiency of Related Work

Here we comment on the performance of the DO-protocol and the BDOZ-protocol, both of which use a very different approach than our protocol and the DN-protocol.

DO-protocol. In the DO-protocol, the computational load is split between a pre-processing and an on-line phase [DO10a]. Their protocol optimizes the cost of the on-line phase, at the expense of the pre-processing phase. The crux of the pre-processing phase is to generate so-called triplets of commitments to random values (a, b, c), where $c = a \cdot b$. One triplet is required per multiplication

[7] In practice, exponentiations modulo $\mathfrak{p}$ are only a few percent slower than modulo n.

[8] The computer used for the benchmarks had an Intel i7 Q820 processor clocked at 1.73 GHz. We used version 5.0.2 of the GNU Multiple Precision Arithmetic Library.

gate. Instead of using traditional zero-knowledge proofs, they uses a technique called "cut-and-choose", where $\mathcal{P}$ generates a number of triplets without proof, and then selectively reveals a fraction of these to $\mathcal{Q}$. Afterwards, $\mathcal{P}$ and $\mathcal{Q}$ "distill" the remaining triplets—which involves interpolation with Lagrange polynomials—to obtain UC-secure triplets.

Their approach however suffers from two drawbacks: *1)* a large number of triplets have to be generated no matter what, and then used up during "distillation" to ensure security, and *2)* due to the Lagrange interpolation, the runtime of the pre-processing phase is *quadratic* in the number of triplets generated in each batch.

In our analysis in Table 1, we used the parameters $\lambda = 0.25$ and $B = 3.6 \cdot \mathsf{s}$ as recommended. We computed the *amortized* runtime (offline + on-line) per muliplication gate when doing exactly $4.8 \cdot \mathsf{s}$ multiplications (the value $4.8 \cdot \mathsf{s}$ was chosen because it comes to within 1% of the minimum runtime per gate for security levels $\mathsf{s} = 80$, $\mathsf{s} = 96$, and $\mathsf{s} = 112$). When more multiplications gates are required, the pre-processing phase should be done in batches of $4.8 \cdot \mathsf{s}$. Note that no matter how many triplets are generated, the pre-processing phase is very slow—at least four minutes for $\mathsf{s} = 80$.

BDOZ-protocol. Similarily to the DO-protocol, in the BDOZ-protocol the computational load is split between a heavy pre-processing phase and a very fast on-line phase (with essentially no cryptographic operations) [BDOZ11]. Like in the DO-protocol, a number of triplets are generated during the pre-processing phase. The technique used to generate them is somewhat different, and as observed by the authors is slightly slower: triplets are generated in batches of s, and a Σ-protocol (with binary challenge run on s instances simultaneously) ensures correctness.

In our analysis in Table 1, we determined the *amortized* runtime per multiplication gate in the pre-processing phase. The online phase was not considered, since it only consists of modular additions and multiplications.

8 Example of a Useful Protocol Constructed with $\mathcal{F}_{\text{ABB}}$

In this section we give an example of how to use our framework to construct a UC-secure variant of the Dodis-Yampolskiy oblivious pseudorandom function [DY05] in a group of order n as originally proposed by Jarecki and Liu [JL09]. Jarecki and Liu proposed a two-party protocol for computing the following oblivious pseudorandom function (OPRF) [JL09], inspired by a similar construct by Dodis and Yampolskiy [DY05]:

$$f_y(x) = \begin{cases} \mathsf{g}^{1/(y+x)} & \text{if } \gcd(y+x, \mathsf{n}) = 1 \\ 1 & \text{otherwise} \end{cases}$$

Where $\mathcal{P}$'s private input is x, $\mathcal{Q}$'s private input is y, $\mathcal{P}$'s output is $f_y(x)$ and $\mathcal{Q}$'s receives a bit b where $b = 0$ iff $\gcd(y + x, \mathsf{n}) = 1$.

Their protocol is only secure against *static* corruptions and has not been proven to be UC secure, which is unfortunate since many of the applications they proposed in their paper would benefit from being able to treat the OPRF generation protocol as a black box. We remedy to this situation here by leveraging $\mathcal{F}_{\text{ABB}}$ and the extensions presented in Section 5. The price to pay is that our construction is about 3.2 times slower (see Table 2).

8.1 Ideal Functionality

For completeness we explicitly show here the ideal functionality $\mathcal{F}_{\text{OPRF}}$ which is parametrized by an abelian group $\mathbb{G}$ of order $\mathtt{n}$ (written multiplicatively), and a genrator $\mathtt{g}$ of $\mathbb{G}$.

- Preparing f_y (needs to be done once only):
  - $\mathtt{input{:}y}$: Accept $\langle \mathtt{input{:}y}, y\rangle$ from $\mathcal{Q}$ where $y \in \mathbb{Z}_{\mathtt{n}}$. Store y: $\bar{y} \leftarrow y$. Send $\langle \mathtt{input{:}y}\rangle$ to $\mathcal{A}$.
  - $\mathtt{ready{:}y}$: Accept $\langle \mathtt{ready{:}y}\rangle$ from $\mathcal{P}$. Send $\langle \mathtt{ready{:}y}\rangle$ to $\mathcal{A}$.
  - $\mathtt{commit{:}y}$ $[\mathtt{input{:}y} \wedge \mathtt{ready{:}y}]$: Accept $\langle \mathtt{commit{:}y}\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{commit{:}y}\rangle$ to $\mathcal{A}$.
  - $\mathtt{done{:}y}$ $[\mathtt{commit{:}y}]$: Accept $\langle \mathtt{done{:}y}\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{done{:}y}\rangle$ to $\mathcal{Q}$.
  - $\mathtt{deliver{:}y}$ $[\mathtt{commit{:}y}]$: Accept $\langle \mathtt{deliver{:}y}\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{deliver{:}y}, \mathtt{g}^{\bar{y}}\rangle$ to $\mathcal{P}$.
- Computing $f_y(x_i)$ (can be repeated many times by using a different $\varphi \in \Sigma^\star$):
  - $\mathtt{input{:}x{:}}\varphi$ $[\mathtt{deliver{:}y}]$: Accept $\langle \mathtt{input{:}x{:}}\varphi, x_\varphi\rangle$ from $\mathcal{Q}$ where $x_\varphi \in \mathbb{Z}_{\mathtt{n}}$. Store x_φ: $\bar{x}_\varphi \leftarrow x_\varphi$. Send $\langle \mathtt{input{:}x{:}}\varphi\rangle$ to $\mathcal{A}$.
  - $\mathtt{ready{:}x{:}}\varphi$ $[\mathtt{done{:}y}]$: Accept $\langle \mathtt{ready{:}x{:}}\varphi\rangle$ from $\mathcal{P}$. Send $\langle \mathtt{ready{:}x{:}}\varphi\rangle$ to $\mathcal{A}$.
  - $\mathtt{lock{:}x{:}}\varphi$ $[\mathtt{input{:}x{:}}\varphi \wedge \mathtt{ready{:}x{:}}\varphi]$: Accept $\langle \mathtt{lock{:}x{:}}\varphi\rangle$ from $\mathcal{A}$. Let $\bar{b}_\varphi \leftarrow 0$ if $\gcd(\mathtt{n}, \bar{y} + \bar{x}_\varphi) = 1$, else $\bar{b}_\varphi \leftarrow 1$. Send $\langle \mathtt{lock{:}x{:}}\varphi\rangle$ to $\mathcal{A}$.
  - $\mathtt{done{:}x{:}}\varphi$ $[\mathtt{lock{:}x{:}}\varphi]$: Accept $\langle \mathtt{done{:}x{:}}\varphi\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{done{:}x{:}}\varphi, \bar{b}_\varphi\rangle$ to $\mathcal{Q}$.
  - $\mathtt{deliver{:}x{:}}\varphi$ $[\mathtt{lock{:}x{:}}\varphi]$: Accept $\langle \mathtt{deliver{:}x{:}}\varphi\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{deliver{:}x{:}}\varphi, f_{\bar{y}}(\bar{x}_\varphi)\rangle$ to $\mathcal{P}$.
- Dealing with corruptions:
  - $\mathtt{corrupt{:}P}$: Accept special $\langle \mathtt{corrupt}\rangle$ message from $\mathcal{P}$. Send $\langle \mathtt{corrupt{:}P}\rangle$ to $\mathcal{A}$.
  - $\mathtt{corrupt{:}Q}$: Accept special $\langle \mathtt{corrupt}\rangle$ message from $\mathcal{Q}$. Send $\langle \mathtt{corrupt{:}Q}\rangle$ to $\mathcal{A}$.
  - $\mathtt{expose{:}y}$ $[\mathtt{input{:}y} \wedge \mathtt{corrupt{:}Q}]$: Accept $\langle \mathtt{expose{:}y}\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{expose{:}y}, \bar{y}\rangle$ to $\mathcal{A}$.
  - $\mathtt{reset{:}y}$ $[\neg\mathtt{commit{:}y} \wedge \mathtt{corrupt{:}Q}]$: Accept $\langle \mathtt{reset{:}y}, y\rangle$ from $\mathcal{A}$. Change y: $\bar{y} \leftarrow y$. Send $\langle \mathtt{reset{:}y}\rangle$ to $\mathcal{A}$.
  - $\mathtt{expose{:}x{:}}\varphi$ $[\mathtt{input{:}x{:}}\varphi \wedge \mathtt{corrupt{:}P}]$: Accept $\langle \mathtt{expose{:}x}\rangle$ from $\mathcal{A}$. Send $\langle \mathtt{expose{:}x}, \bar{x}_\varphi\rangle$ to $\mathcal{A}$.
  - $\mathtt{reset{:}x{:}}\varphi$ $[\neg\mathtt{lock{:}x{:}}\varphi \wedge \mathtt{corrupt{:}Q}]$: Accept $\langle \mathtt{reset{:}x{:}}\varphi, x_\varphi\rangle$ from $\mathcal{A}$. Change x_φ: $\bar{x}_\varphi \leftarrow x_\varphi$. Send $\langle \mathtt{reset{:}x{:}}\varphi\rangle$ to $\mathcal{A}$.

8.2 Construction

Preparing f_y (needs to be done once only):

1. $\mathcal{Q}$ inputs value y to $\mathcal{F}_{\text{ABB}}$ with identifier k_0 using the *Input* instruction.
2. $\mathcal{Q}$ outputs the "public key" $\mathtt{g}^y$ using the *Exponentiated Output* instruction. (This step can be omitted if the value $\mathtt{g}^y$ is not needed. Indeed $\mathcal{Q}$ is committed to y through $\mathcal{F}_{\text{ABB}}$ anyway.)

Computing $f_y(x_i)$ (can be repeated many times):

1. $\mathcal{P}$ inputs value x_i to $\mathcal{F}_{\text{ABB}}$ with identifier k_{3i+1} using the *Input* instruction.
2. They compute $y + x_i$: $V[k_{3i+2}] \leftarrow V[k_0] + V[k_{3i+1}]$.
3. They invert the previous result using the protocol shown in Section 5.1: $V[k_{3i+3}] \leftarrow (V[k_{3i+2}])^{-1}$. (If the inversion fails, then $\mathcal{P}$ and $\mathcal{Q}$ output 1 and skip the next step—this is similar to how Jarecki et al. proceed [JL09]).
4. $\mathcal{P}$ retrieves $\mathtt{g}^{V[k_{3i+3}]} = \mathtt{g}^{1/(y+x_i)} = f_y(x_i)$ using *Exponentiated Output*.
5. $\mathcal{Q}$ returns 0.

	Runtime for OPRF setup and one OPRF compute	with $\mathsf{s} = 80$
This work	$(219 \cdot \mathsf{s} + 290 \cdot \mathrm{lb}\,\mathsf{n}) \cdot \exp.\mathsf{n} + (74 \cdot \mathsf{s} + 52.5 \cdot \mathrm{lb}\,\mathsf{n}) \cdot \exp.\mathsf{n}^2$	836 ms
Jarecki-Liu [JL09]	$(\ 45 \cdot \mathsf{s} + \ \ 8 \cdot \mathrm{lb}\,\mathsf{n}) \cdot \exp.\mathsf{n} + (14 \cdot \mathsf{s} + 40 \ \ \cdot \mathrm{lb}\,\mathsf{n}) \cdot \exp.\mathsf{n}^2$	263 ms

Table 2. Estimated runtime per OPRF computation including preparation, using the same notation as Table 1. Note that Jarecki and Liu's protocol [JL09] is not UC-secure, and only secure against *static* corruptions.

8.3 Security

Correctness and privacy of the input follow directly from the construction of the extended $\mathcal{F}_{\mathrm{ABB}}$.

Acknowledgements

We are grateful to Stephan Krenn and to the anonymous reviewers for their comments. This work was supported by the European Community through the Seventh Framework Programme (FP7), under grant agreements n°257782 for the project ABC4Trust and n°321310 for the project PERCY.

References

[BCC+11] S. Babbage, D. Catalano, C. Cid, B. de Weger, O. Dunkelman, C. Gehrmann, L. Granboulan, T. Güneysu, J. Hermans, T. Lange, A. Lenstra, C. Mitchell, M. Näslund, P. Nguyen, C. Paar, K. Paterson, J. Pelzl, T. Pornin, B. Preneel, C. Rechberger, V. Rijmen, M. Robshaw, A. Rupp, M. Schläffer, S. Vaudenay, F. Vercauteren, and M. Ward. ECRYPT II Yearly Report on Algorithms and Keysizes, 2011.

[BDOZ11] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT*, pages 169–188, 2011.

[BIB89] J. Bar-Ilan and D. Beaver. Non-Cryptographic Fault-Tolerant Computing in Constant Number of Rounds of Interaction. In *PODC*, pages 201–209, 1989.

[Can00] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000.

[CCGS10] J. Camenisch, N. Casati, T. Groß, and V. Shoup. Credential Authenticated Identification and Key Exchange. In *CRYPTO*, pages 255–276, 2010.

[CDN01] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *EUROCRYPT*, pages 280–299, 2001.

[CES13] J. Camenisch, R. R. Enderlein, and V. Shoup. Practical Universally Composable Circuit Evaluation over Z_n. *IACR Cryptology ePrint Archive*, 2013:205, 2013.

[CF01] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO*, pages 19–40, 2001.

[CHK05] R. Canetti, S. Halevi, and J. Katz. Adaptively-Secure, Non-interactive Public-Key Encryption. In *TCC*, pages 150–168, 2005.

[CKL06] R. Canetti, E. Kushilevitz, and Y. Lindell. On the Limitations of Universally Composable Two-Party Computation Without Set-Up Assumptions. *J. Cryptology*, 19(2):135–167, 2006.

[CKS11] J. Camenisch, S. Krenn, and V. Shoup. A Framework for Practical Universally Composable Zero-Knowledge Protocols. In *ASIACRYPT*, pages 449–467, 2011.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. In *STOC*, pages 494–503, 2002.

[CS97] J. Camenisch and M. Stadler. Proof Systems for General Statements about Discrete Logarithms. *Institute for Theoretical Computer Science, ETH Zürich, Tech. Rep.*, 260, 1997.

[CS03] J. Camenisch and V. Shoup. Practical Verifiable Encryption and Decryption of Discrete Logarithms. In *CRYPTO*, pages 126–144, 2003.

[DFK+06] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC*, pages 285–304, 2006.

[DJ03] I. Damgård and M. Jurik. A Length-Flexible Threshold Cryptosystem with Applications. In *ACISP*, pages 350–364, 2003.

[DK10] I. Damgård and M. Keller. Secure Multiparty AES. In *Financial Cryptography*, pages 367–374, 2010.

[DKL+12] I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol. In *SCN*, pages 241–263, 2012.

[DKL+13] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical Covertly Secure MPC for Dishonest Majority - or: Breaking the SPDZ Limits. *ESORICS*, 2013.

[DN03] I. Damgård and J. B. Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *CRYPTO*, pages 247–264, 2003.

[DO10a] I. Damgård and C. Orlandi. Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost. In *CRYPTO*, pages 558–576, 2010.

[DO10b] I. Damgård and C. Orlandi. Multiparty Computation for Dishonest Majority: from Passive to Active Security at Low Cost. *IACR Cryptology ePrint Archive*, 2010:318, 2010.

[DPSZ12] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*, pages 643–662, 2012.

[DY05] Y. Dodis and A. Yampolskiy. A Verifiable Random Function with Short Proofs and Keys. In *Public Key Cryptography*, pages 416–431, 2005.

[HS11] D. Hofheinz and V. Shoup. GNUC: A New Universal Composability Framework. *IACR Cryptology ePrint Archive*, 2011:303, 2011.

[IPS08] Y. Ishai, M. Prabhakaran, and A. Sahai. Secure Arithmetic Computation with No Honest Majority. *IACR Cryptology ePrint Archive*, 2008:465, 2008.

[IPS09] Y. Ishai, M. Prabhakaran, and A. Sahai. Secure Arithmetic Computation with No Honest Majority. In *TCC*, pages 294–314, 2009.

[JL09] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC*, pages 577–594, 2009.

[JS07] S. Jarecki and V. Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. In *EUROCRYPT*, pages 97–114, 2007.

[KSS12] B. Kreuter, A. Shelat, and C. Shen. Towards Billion-Gate Secure Computation with Malicious Adversaries. *IACR Cryptology ePrint Archive*, 2012:179, 2012.

[Küs06] R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *IEEE Computer Security Foundations Workshop*, pages 309–320, 2006.

[LP07] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *EUROCRYPT*, pages 52–78, 2007.

[LPS08] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In *SCN*, pages 2–20, 2008.

[MR11] U. Maurer and R. Renner. Abstract Cryptography. In *ICS*, pages 1–21, 2011.

[Nie03] J. B. Nielsen. *On Protocol Security in the Cryptographic Model*. PhD thesis, BRICS, Computer Science Department, University of Aarhus, 2003.

[NNOB12] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO*, pages 681–700, 2012.

[NO09] J. B. Nielsen and C. Orlandi. LEGO for Two-Party Secure Computation. In *TCC*, pages 368–386, 2009.

[Pai99] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*, pages 223–238, 1999.

[PSSW09] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, pages 250–267, 2009.

[PW01] B. Pfitzmann and M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *IEEE Security & Privacy*, pages 184–200, 2001.

[Sho04] V. Shoup. Sequences of Games: a Tool for Taming Complexity in Security Proofs. *IACR Cryptology ePrint Archive*, 2004:332, 2004.

[WJ79] S. S. Wagstaff Jr. Greatest of the Least Primes in Arithmetic Progressions Having a Given Modulus. *Mathematics of Computation*, 33(147):pp. 1073–1080, 1979.

A $\mathcal{F}_{\mathbf{ZK}}$ and $\mathcal{F}_{\mathbf{gZK}}$ Ideal Functionalities

In this Section, we show the ideal functionalities for zero-knowledge proofs of knowledge and existance: $\mathcal{F}_{\text{ZK}}$ and $\mathcal{F}_{\text{gZK}}$, respectively.

A.1 $\mathcal{F}_{\mathrm{ZK}}$

The following is the formal definition of the $\mathcal{F}_{\mathrm{ZK}}$ functionality for universally composable zero-knowledge proofs of knowledge in the GNUC model [HS11]. It is designed to be used in a setting where dynamic corruption with erasures are allowed. This ideal functionality is parametrized by a binary predicate x and a leakage function ℓ.

- `send` : Accept $\langle \texttt{send}, x, w \rangle$ from $\mathcal{P}$ where $R(x, w) = 1$. Store the instance and witness: $\bar{x} \leftarrow x$ and $\bar{w} \leftarrow w$. Send $\langle \texttt{send}, \ell(x, w) \rangle$ to $\mathcal{A}$.
- `ready` : Accept $\langle \texttt{ready} \rangle$ from $\mathcal{Q}$. Send $\langle \texttt{ready} \rangle$ to $\mathcal{A}$.
- `lock` [`send` $\wedge$ `ready`] : Accept $\langle \texttt{lock} \rangle$ from $\mathcal{A}$. Send $\langle \rangle$ to $\mathcal{A}$.
- `done` [`lock`] : Accept $\langle \texttt{done} \rangle$ from $\mathcal{A}$. Send $\langle \texttt{done} \rangle$ to $\mathcal{P}$.
- `deliver` [`lock`] : Accept $\langle \texttt{deliver}, L \rangle$ from $\mathcal{A}$, where $L = \ell(\bar{x}, \bar{w}) \vee$ [`corrupt:Q`]. Send $\langle \texttt{deliver}, \bar{x} \rangle$ to $\mathcal{Q}$.
- `corrupt:P` : Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{P}$. Send $\langle \texttt{corrupt:P} \rangle$ to $\mathcal{A}$ together with an invitation for the message $\langle \texttt{expose} \rangle$.
- `corrupt:Q` : Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{Q}$. Send $\langle \texttt{corrupt:Q} \rangle$ to $\mathcal{A}$.
- `reset` [$\neg$`lock` $\wedge$ `corrupt:P`] : Accept $\langle \texttt{reset}, x, w \rangle$ from $\mathcal{A}$, where $R(x, w) = 1$. Store the instance and witness: $\bar{x} \leftarrow x$ and $\bar{w} \leftarrow w$. Send $\langle \rangle$ to $\mathcal{A}$.
- `expose` [`send` $\wedge$ $\neg$`lock` $\wedge$ `corrupt:P`] : Accept $\langle \texttt{expose} \rangle$ from $\mathcal{A}$. Send $\langle \texttt{expose}, \bar{x}, \bar{w} \rangle$ to $\mathcal{A}$.

A.2 $\mathcal{F}_{\mathrm{gZK}}$

The functionality $\mathcal{F}_{\mathrm{gZK}}$ is a tool which allows us to simplify the security proof of protocols which use zero-knowledge proofs of *existence*. This functionality was proposed by Camenisch et al. [CKS11]. The two major differences between $\mathcal{F}_{\mathrm{ZK}}$ and $\mathcal{F}_{\mathrm{gZK}}$ is that the latter: *1)* does not check its inputs, and *2)* does not allow the adversary to extract the witnesses quantified by $\exists$.

One must be careful with this functionality, since it is not an ideal functionality in the UC sense. Indeed the functionality is quite useless by itself. By using a special composition theorem by Camenisch et al., one can prove that if the $\mathcal{F}_{\mathrm{gZK}}$-hybrid protocol is secure against a weak class of environments called *nice* environments, then the $\mathcal{F}_{\mathrm{sch}}$-hybrid protocol in which all instances of $\mathcal{F}_{\mathrm{gZK}}$ have been replaced by a specific zero-knowledge protocol is secure in the UC-sense.

Like $\mathcal{F}_{\mathrm{ZK}}$, $\mathcal{F}_{\mathrm{gZK}}$ is designed to be used in a setting where dynamic corruption with erasures are allowed; and $\mathcal{F}_{\mathrm{gZK}}$ is parametrized by a binary predicate x and a leakage function ℓ.

- `send` : Accept $\langle \texttt{send}, x, w_k, w_e \rangle$ from $\mathcal{P}$. Store the instance and all witnesses quantified by $\exists$: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. The ideal functionality $\mathcal{F}_{\mathrm{gZK}}$, being *gullible*, does not check if the predicate holds. Send $\langle \texttt{send}, \ell(x, w_k) \rangle$ to $\mathcal{A}$.
- `ready` : Accept $\langle \texttt{ready} \rangle$ from $\mathcal{Q}$. Send $\langle \texttt{ready} \rangle$ to $\mathcal{A}$.
- `lock` [`send` $\wedge$ `ready`] : Accept $\langle \texttt{lock} \rangle$ from $\mathcal{A}$. Send $\langle \rangle$ to $\mathcal{A}$.
- `done` [`lock`] : Accept $\langle \texttt{done} \rangle$ from $\mathcal{A}$. Send $\langle \texttt{done} \rangle$ to $\mathcal{P}$.
- `deliver` [`lock`] : Accept $\langle \texttt{deliver}, L \rangle$ from $\mathcal{A}$, where $L = \ell(\bar{x}, \bar{w}_k) \vee$ [`corrupt:Q`]. Send $\langle \texttt{deliver}, \bar{x} \rangle$ to $\mathcal{Q}$.
- `corrupt:P` : Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{P}$. Send $\langle \texttt{corrupt:P} \rangle$ to $\mathcal{A}$ together with an invitation for the message $\langle \texttt{expose} \rangle$.
- `corrupt:Q` : Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{Q}$. Send $\langle \texttt{corrupt:Q} \rangle$ to $\mathcal{A}$.

– `reset` [$\neg$`lock` $\wedge$ `corrupt`:P] : Accept $\langle$`reset`$, x, w_k, w_e\rangle$ from $\mathcal{A}$. The ideal functionality $\mathcal{F}_{\text{gZK}}$, being *gullible*, does not check if the predicate holds. Store the instance and all witnesses quantified by $\forall$: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. Send $\langle\rangle$ to $\mathcal{A}$.
– `expose` [`send` $\wedge$ $\neg$`lock` $\wedge$ `corrupt`:P] : Accept $\langle$`expose`$\rangle$ from $\mathcal{A}$. Send $\langle$`expose`$, \bar{x}, \bar{w}_k\rangle$ to $\mathcal{A}$.