Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique[*]

(Technical report – September 11, 2013)

Luís T. A. N. Brandão[†]

University of Lisbon
Faculty of Sciences / LaSIGE
Lisboa, PORTUGAL
lbrandao@fc.ul.pt

Carnegie Mellon University
Electrical & Computer Engineering
Pittsburgh, USA
lbrandao@cmu.edu

Abstract. A *secure two-party computation* (S2PC) protocol allows two parties to compute over their combined private inputs, as if intermediated by a trusted third party. In the active model, security is maintained even if one party is malicious, deviating from the protocol specification. For example, a honest party retains *privacy* of its input and is ensured a *correct* output. This can be achieved with a *cut-and-choose* of *garbled circuits* (C&C-GCs), where some GCs are *verified* for correctness and the remaining are *evaluated* to determine the circuit output. This paper presents a new C&C-GCs-based S2PC protocol, with significant advantages in efficiency and applicability. First, in contrast with prior protocols that require a majority of *evaluated* GCs to be correct, the new protocol only requires that at least one *evaluated* GC is correct. In practice this reduces the total number of GCs to approximately one third, for the same statistical security goal. This is accomplished by augmenting the C&C with a new *forge-and-lose* technique based on bit commitments with trapdoor. Second, the output of the new protocol includes reusable XOR-homomorphic bit commitments of all circuit input and output bits, thereby enabling efficient linkage of several S2PCs in a reactive manner. The protocol has additional interesting characteristics (which may allow new comparison tradeoffs). The number of exponentiations is only linear with the number of input and output wires and a statistical parameter – this is an improvement over protocols whose number of exponentiations is proportional to the number of GCs multiplied by the number of input and output wires. It uses unconditionally hiding bit commitments with trapdoor as the basis of oblivious transfers, with the circuit evaluator choosing a single value and the circuit constructor receiving two (a sort of 2-out-of-1 oblivious transfer, instead of the typical 1-out-of-2). The verification of consistency of circuit input and output keys across different GCs is embedded in the C&C structure.

Keywords: secure two-party computation, cut-and-choose, garbled circuits, forge-and-lose, homomorphic bit-commitments with trapdoor.

[*] This is the full version. An extended abstract will appear at ASIACRYPT 2013 and be published by Springer-Verlag.

[†] Support for this research was provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under Grant SFRH/BD/33770/2009, and through the Multiannual Funding Programme (LASIGE), while the author was a Ph.D. student at FCUL-DI-LaSIGE and CMU-ECE. Alternative email contact: luis.papers@gmail.com.

Index

1 Introduction

Secure two-party computation is a general cryptographic functionality that allows two parties to interact as if intermediated by a *trusted third party* [Gol04]. A canonical example is *the millionaire's problem* [Yao82], where two parties find who is the richer of the two, without revealing to the other any additional information about the amounts they own. Applications of secure computation can be envisioned in many cases where mutually distrustful parties can benefit from learning something from their combined data, without sharing their inputs [Kol09]. For example, two parties may evaluate a data mining algorithm over their combined databases, in a privacy-preserving manner [LP02]. On a different example, one party with a private message may obtain a respective message authentication code calculated with a secret key from another party (i.e., a blind MAC) [PSSW09]. This paper considers secure two-party evaluation of Boolean circuits, henceforth denoted "S2PC", which can be used to solve the mentioned examples. Each party begins the interaction with a private input encoded as a bit-string, and a public specification of a Boolean circuit that computes an intended function. Then, the two parties interact so that each party learns only the output of the respective circuit evaluated over both private inputs. Probabilistic functionalities can be implemented by letting the two parties hold additional random bits as part of their inputs.

This paper focuses on the *malicious model*, where parties might maliciously deviate from the protocol specification in a computationally bounded way. Furthermore, within the *standard model* of cryptography, adopted herein, it is assumed that some problems are computationally intractable, such as those related with inverting trapdoor permutations. Security is defined within the ideal/real simulation paradigm [Can00]; i.e., a protocol is said to implement S2PC if it *emulates* an ideal functionality where a trusted third party mediates the communication and computation between the two parties. The trusted party receives the private inputs from both parties, makes the intended computation locally and then delivers the final private outputs to the respective parties.

As a starting point, this paper considers the *cut-and-choose* (C&C) of *garbled circuits* (GCs) approach to achieve S2PC. Here, a circuit constructor party ($\mathrm{P_A}$) builds several GCs (cryptographic versions of the Boolean circuit that computes the intended function), and then the other party, the circuit evaluator ($\mathrm{P_B}$), verifies some GCs for correctness and evaluates the remaining to obtain the information necessary to finally decide a correct circuit output. Recently, this approach has had the best reported efficiency benchmark [KSS12; FN13] for S2PC protocols with a constant number of rounds of communication.

1.1 Contributions

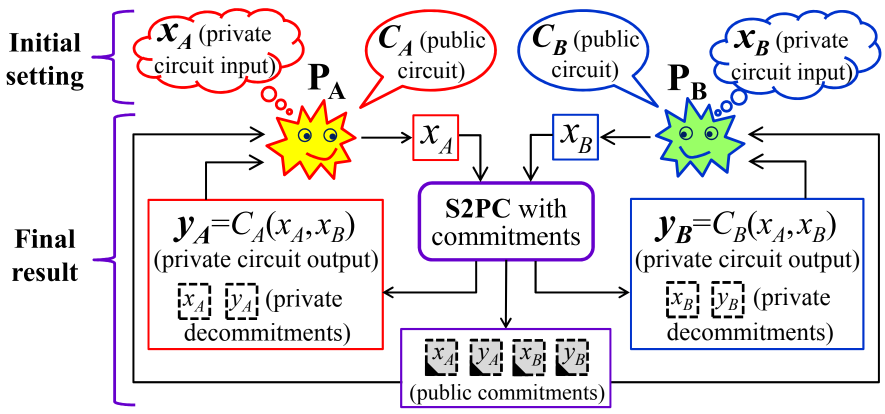
This paper introduces a new *bit commitment* (BitCom) approach and a new evaluation technique, dubbed *forge-and-lose*, and blends them into a C&C approach, to achieve a new C&C-GCs-based S2PC protocol with significant improvements in applicability and efficiency.

Applicability. The new protocol achieves S2PC-with-BitComs, as illustrated in Fig. 1. Specifically, both parties receive random BitComs of all circuit input and output bits, with each party also learning the decommitments of only her respective circuit input and output bits. This is an augmented version of secure circuit evaluation. Given the reusability of BitComs, the protocol can be taken as a building block to achieve other goals, such as reactive linkage of several S2PCs, efficiently and securely linking the input and output bits of one execution with the input bits of subsequent executions. Furthermore, given the XOR-homomorphic properties of these BitComs, a party may use efficient specialized *zero-knowledge proof*s (ZKPs) to prove that her private input bits in one execution satisfy certain non-deterministic polynomially verifiable relations with the private input and output bits of previous executions.[1] In previous C&C-GCs-based solutions, without committed inputs and outputs with homomorphic properties, such general linkage would be conceivable but using more expensive ZKPs of correct behavior.

The main technical description in this paper is focused on a standalone 1-output protocol, where the two parties, $\mathrm{P_A}$ and $\mathrm{P_B}$, interact so that only $\mathrm{P_B}$ learns a circuit output.[2] In the new BitCom approach, the two possible decommitments of the BitCom of each circuit input or output bit (independent of the number of GCs) are connected to the two keys of the respective input or output wire of each GC, via a new construction dubbed *connector*. $\mathrm{P_A}$ commits to these *connectors* and then reveals them partially for *verification* or *evaluation*. This ensures, within the C&C, the correctness of circuit input keys and the privacy of decommitments of BitComs, without requiring additional ZKPs.

[1] For simplicity, "ZKPs" is used hereafter both for ZK *proofs* and for ZK *arguments*.

[2] The "1-output" characterization refers to only one party learning a circuit output, though in rigor the protocol implements a probabilistic 2-output functionality (as both parties receive random BitComs).


Initial setting
Final result
x_A (private circuit input)
C_A (public circuit)
C_B (public circuit)
x_B (private circuit input)
P_A
P_B
x_A
x_B
S2PC with commitments
$y_A = C_A(x_A, x_B)$ (private circuit output)
x_A y_A (private decommitments)
$y_B = C_B(x_A, x_B)$ (private circuit output)
x_B y_B (private decommitments)
x_A y_A x_B y_B (public commitments)

Legend: $\mathrm{P_A}$ and $\mathrm{P_B}$ (the names of the two parties); x_p, y_p and C_p (the private circuit input, the private circuit output and the public circuit specification of party P_p, respectively, with p being A or B); ▨ and ⬚ (commitment and decommitment, respectively, of the variable inscribed inside the dashed square). Colors red, blue and purple are related with $\mathrm{P_A}$, $\mathrm{P_B}$ and both parties, respectively.

Fig. 1: **Secure Two-Party Computation with Committed Inputs and Outputs**

The BitCom approach enables particularly efficient extensions of this 1-output protocol into 2-output protocols where both parties learn a respective private circuit-output.

Efficiency. The new protocol requires only an optimal minimum number of GCs in the C&C, for a certain soundness guarantee (i.e., for an upper bound on the probability with which a malicious $\mathrm{P_A}$ can make $\mathrm{P_B}$ accept an incorrect output). Specifically, by only requiring that at least one evaluation GC is correct, the total number of GCs is reduced asymptotically about 3.1 times, in comparison with the previously best known C&C-GCs configuration [SS11] that required a correct majority of evaluation GCs. The significance of this improvement stems from the number of GCs being the source of most significant cost of C&C-GCs-based S2PC protocols, for circuits of practical size. **Remark:** two different techniques [Lin13; HKE13] developed in concurrent research also just require a single evaluation GC to be correct – a brief comparison is made in §7.1, but the remaining introductory part of this paper only discusses the typical C&C-GCs approach that requires a correct majority of evaluation GCs.

The reduction in number of GCs is achieved via a new forge-and-lose technique, providing a path by which $\mathrm{P_B}$ can recover the correct final output when there are inconsistent outputs in the evaluated GCs. Assume that $\mathrm{P_A}$ is able to *forge* a GC; i.e., build an incorrect GC that, if selected for *evaluation*, degarbles smoothly into an output that cannot be perceived as incorrect. Then, $\mathrm{P_B}$ somehow combines the forged output with a correct output, in a way that reveals a secret key (a trapdoor) with which the input of $\mathrm{P_A}$ has previously been encrypted (committed). In this way, $\mathrm{P_A}$ *loses* privacy of her input bits, enabling $\mathrm{P_B}$ to compute the intended circuit output in the clear.

The protocol can be easily adjusted to integrate several optimizations in communication and memory, such as *random seed checking* [GMS08] and *pipelining* [HEKM11]. Since the garbling scheme is abstracted, the protocol is also compatible with many garbling optimizations, e.g., *point and permute* [NPS99], *XOR for free* [KS08b], *garbled row reduction* [PSSW09], *dual-key cipher* [BHR12].

1.2 Roadmap

The remainder of this paper is organized as follows. Section 2 reviews the basic building blocks of the typical C&C-GCs approach and some properties of BitCom schemes. Section 3 introduces a new BitCom approach, explaining how BitComs can be *connected* to circuit input and output wire keys, to ensure the consistency of the keys across different GCs. Section 4 describes the forge-and-lose technique, achieving a major efficiency improvement over the typical cut-and-choose approach. Section 5 presents the new protocol for 1-output S2PC-with-BitComs, where only one party learns a private circuit-output, and both parties learn BitComs of the input and output bits of both parties. Section 6 comments on the complexity of the protocol and shows how the BitCom approach enables efficient linkage of S2PCs. Section 7 compares some aspects of related work. The Appendix includes a more formal description, analysis and optimization of the protocol and a proof of security.

2 Background

This section reviews some basic notions of the C&C-GCs solution for S2PC (§2.1) and some useful properties of certain BitCom schemes (§2.2).

2.1 C&C-GCs-based S2PC

Basic garbled-circuit approach. The theoretical feasibility of S2PC, for functions efficiently representable by Boolean circuits, was initially shown by Yao [Yao86].[3] In the *semi-honest model* (where parties behave correctly during the protocol) simplified to the 1-output setting, only one of the parties ($\mathrm{P_B}$) intends to learn the output of an agreed Boolean circuit that computes the desired function. The *basic GC approach* starts with the other party ($\mathrm{P_A}$) building a GC – a cryptographic version of the Boolean circuit, which evaluates keys (e.g., random bit-strings) instead of clear bits. The GC is a directed acyclic graph of garbled gates, each receiving keys as input and outputting new keys. Each gate output key has a corresponding underlying bit (the result of applying the Boolean gate operation to the bits underlying the corresponding input keys), but the bit correspondence is hidden from $\mathrm{P_B}$. $\mathrm{P_A}$ sends the GC and one circuit input key per each input wire to $\mathrm{P_B}$. Then, $\mathrm{P_B}$ obliviously evaluates the GC, learning only one key per intermediate wire but not the respective underlying bit. Finally, each circuit output bit is revealed by a special association with the key learned for the respective circuit output wire. Lindell and Pinkas [LP09] prove the security of a version of Yao's protocol (valid for a 2-output setting).

There are many known proposals for garbling schemes [BHR12]. This paper abstracts from specific constructions, except for making the typical assumptions that: (i) with two valid keys per circuit input wire (and possibly some additional randomness used to generate the GC), $\mathrm{P_B}$ can *verify the correctness* of the GC, in association with the intended Boolean circuit, and determine the bit underlying each input and output key; and (ii) with a single key per circuit input wire, $\mathrm{P_B}$ can *evaluate* the GC, learning the bits corresponding to the obtained circuit output keys, but not learn additional information about the bit underlying the single key obtained for each input wire of $\mathrm{P_A}$ and for each intermediate wire.

Oblivious transfer. An essential step of the basic GC-based protocol requires, for each circuit input wire of $\mathrm{P_B}$ (the GC-evaluator), that $\mathrm{P_A}$ (the GC-constructor) sends to $\mathrm{P_B}$ the key corresponding to the respective input bit of $\mathrm{P_B}$, but without $\mathrm{P_A}$ learning

[3] See [BHR12, §1] for a brief historical account of the origin of the garbled-circuit approach, including references to [GMW87; BMR90; NPS99].

what is the bit value. This is typically achieved with 1-out-of-2 *oblivious transfer*s (OTs) [Rab81; EGL85; NP01], where the sender ($\mathrm{P_A}$) selects two keys per wire, but the receiver ($\mathrm{P_B}$) only learns one of its choice, without the sender learning which one. Some protocols use enhanced variations, e.g., committing OT [CGT95], committed OT [KS06], cut-and-choose OT [LP11], authenticated OT [NNOB12], string-selection OT [KK12]. In practice, the computational cost of OTs is often significant in the overall complexity of protocols, though asymptotically the cost can be amortized with techniques that allow extending a few OTs to a large number of them [Bea96; IKNP03; NNOB12].

The new protocol presented in this paper uses OTs at the BitCom level, to coordinate decommitments between the two parties, as follows. For each circuit input bit of $\mathrm{P_B}$, $\mathrm{P_B}$ selects a bit encoding (a decommitment) and uses it to produce the respective BitCom. Then, $\mathrm{P_A}$ uses a trapdoor to learn *two* decommitments (i.e., bit-encodings for the two bits) for the same BitCom. These OTs are herein dubbed *2-out-of-1 OTs*, since one party chooses *one* value and leads the other party to learn *two* values. This is in contrast with the typical 1-out-of-2 OT (commonly used directly at the level of wire keys), where $\mathrm{P_A}$ chooses *two* keys and leads $\mathrm{P_B}$ to learn *one* of them.

Cut-and-choose approach. Yao's protocol is insecure in the malicious model. For example, a malicious $\mathrm{P_A}$ could construct an undetectably incorrect GC, by changing the Boolean operations underlying the garbled gates, but maintaining the correct graph topology of gates and wires. To solve this, Pinkas [Pin03] proposed a C&C approach, achieving 2-output S2PC via a single-path approach where only $\mathrm{P_B}$ evaluates GCs. A simplified high level description follows. $\mathrm{P_A}$ constructs a set of GCs. $\mathrm{P_B}$ *cuts* the set into two complementary subsets and *chooses* one to *verify* the correctness of the respective GCs. If no problem is found, $\mathrm{P_B}$ *evaluates* the remaining GCs to obtain, from a consistent majority, its own output bits and a masked version of the output of $\mathrm{P_A}$. $\mathrm{P_B}$ sends to $\mathrm{P_A}$ a modified version of the masked output of $\mathrm{P_A}$, without revealing from which GC it was obtained. Finally, $\mathrm{P_A}$ unmasks her final output bits. This approach has two main inherent challenges: (1) how to *ensure that input wire keys are consistent across GCs, such that equivalent input wires receive keys associated with the same input bits (in at least a majority of evaluated GCs)*; (2) how to *guarantee that the modified masked-output of P_A is correct and does not leak private information of P_B*. Progressive solutions proposed across recent years have solved subtle security issues, e.g., the selective-failure-attack [MF06; KS06], and improved the practical efficiency of C&C-GC-based methods [LP07; Woo07; KS08a; NO09; PSSW09; LP11; SS11]. As a third challenge, the number of GCs still remains a primary source of inefficiency, in these solutions that require a correct majority of GCs selected for evaluation. For example, achieving 40 bits of statistical security[4] requires at least 123 GCs (74 of which are for *verification*). Asymptotically, the optimal C&C partition (three fifths of verification GCs) leads to about 0.322 bits of statistical security per GC [SS11].

The BitCom approach developed in this paper deals with all these challenges. First, taking advantage of XOR-homomorphic BitComs, the verification of consistency of input wire keys of both parties is embedded in the C&C, without an ad-hoc ZKP of consistency of keys across different GCs. Second, $\mathrm{P_B}$ can directly learn, from the GC evaluation, decommitments of BitComs of one-time-padded (i.e., masked) output bits of $\mathrm{P_A}$, and then simply send these decommitments to $\mathrm{P_A}$. Privacy is preserved because the decommitments

[4] The number of bits of statistical security is the additive inverse of the logarithm base 2 of the maximum error probability, i.e., for which a malicious $\mathrm{P_A}$ can make $\mathrm{P_B}$ accept an incorrect output.

do not vary with the GC index. Correctness is ensured because the decommitments are verifiable (i.e., authenticated) against the respective BitComs. The BitCom approach also enables achieving 2-output S2PC via a dual-path execution approach – the parties play two 1-output S2PCs, with each party playing once as GC evaluator of only her own intended circuit, using the same BitComs of input bits in both executions.[5] Third, the BitCom approach enables the forge-and-lose technique, which reduces the correctness requirement to only having at least one correct *evaluation* GC, thus increasing the statistical security to about 1 bit per GC (see details in §A).

2.2 Bit Commitments

The BitCom approach introduced in this paper is based on several properties of (some) BitCom schemes, reviewed hereafter. A BitCom scheme [Blu83; BCC88] is a two-party protocol for committing and revealing individual bits. In a *commit* phase, it allows a *sender* to commit to a bit value, by producing and sending a BitCom value to the *receiver*. The BitCom *binds* the *sender* to the chosen bit and, initially, *hides* the bit value from the *receiver*. Then, in a *reveal* phase, the *sender* discloses a private bit-encoding (the decommitment), which allows the *receiver* to learn the committed bit and *verify* its correctness. A scheme is *XOR-homomorphic* if any pair of BitComs can be combined (under some group operation) into a new BitCom that commits the XOR of the original committed bits, and if the same can be done with the respective decommitments.

The following paragraphs describe several properties related with decommitments and trapdoors of practical BitCom schemes. For simplicity, the description focuses on a scheme based on a *square* operation with some useful collision-resistance (i.e., "claw-free" [GMR84; Dam88]) properties.

Unconditionally hiding (UH). A BitCom scheme is called UH if, before the *reveal* phase, a *receiver* with unbounded computational power cannot learn anything about the committed bit. If there is a trapdoor (known by the receiver), then it can be used to retrieve, from any BitCom, respective bit-encodings of both bits. Still, this does not reveal any information about which bit the *sender* might have committed to. A practical instantiation was used by Blum for *coin flipping* [Blu83]. There, in a multiplicative group modulo a Blum integer with factorization unknown by the *sender*, bits 0 and 1 are encoded as group-elements with Jacobi Symbol 1 or -1, respectively.[6] The *commitment* of a bit is achieved by sending the square of a random encoding of the bit. The *revealing* is achieved by sending the known square-root.

Henceforth, a XOR-homomorphic UH BitCom scheme is suggestively dubbed a *2-to-1 square scheme* if it also has the following three useful properties:

– **Proper square-roots.** *Any BitCom (dubbed* square*) has exactly two decommitments (dubbed* proper square-roots*), encoding different bits.* In the Blum integer example,

[5] This is a concrete C&C-GC-based dual-path solution to 2-output S2PC, where the circuits evaluated by each party only compute her respective output. [Kir08, §6.6] and [SS11, §1.2] conceptualized dual-path approaches in high level, but did not explain how to ensure the same input across the two executions. Other dual-path approaches have been proposed using a single GC per party (i.e., not C&C-based), but with potential leakage of one bit of information [MF06; HKE12]. A recent method [HKE13] (see comparison in §7) devised a C&C-based dual-path approach but requiring both parties to evaluate GCs with the same underlying Boolean circuit (for some 2-output functionalities this implies that GCs have the double of the size).

[6] A Blum integer is the product of two prime powers, where each prime is congruent with 3 modulo 4, and each power has an odd exponent. For a fixed Blum integer, the Jacobi Symbol is a completely multiplicative function that maps any group element into 1 or -1 (more detailed theory can be found, for example, in [NZM91]).

each square has four square-roots, two per bit, but it is possible to define a single proper square-root per bit (e.g., the square-root whose least significant bit is equal to the encoded bit). The multiplicative group (set of residues and respective multiplication operation) can be easily adjusted to consider only proper square-roots, since the additive inverse of a non-proper square root is a proper square-root encoding the same bit.

- **From trapdoor to decommitments.** *There is a trapdoor whose knowledge allows extracting a pair of proper square-roots (the two decommitments) from any square (the BitCom).* Such pair is dubbed a *non-trivially correlated pair*, in the sense that the two proper square-roots are related but cannot be simultaneously found (except with the help of a trapdoor). This property allows a 2-out-of-1 OT: $\mathrm{P_B}$ selects a proper square-root and sends its square to $\mathrm{P_A}$, who then uses the trapdoor to obtain the two proper square-roots. In the Blum integer example, the trapdoor is its factorization.
- **From decommitments to trapdoor.** *Any non-trivially correlated pair is a trapdoor.* This is useful for the forge-and-lose technique, as the discovery (by $\mathrm{P_B}$) of such a pair (a trapdoor of $\mathrm{P_A}$), in case $\mathrm{P_A}$ acted maliciously, is the condition that allows $\mathrm{P_B}$ to decrypt the input bits of $\mathrm{P_A}$. In the Blum integer example, its factorization can be found from any pair of proper square-roots of the same square.

Unconditionally binding (UB). A BitCom scheme is called UB if a *sender* with unbounded computational power cannot make the *receiver* accept an incorrect bit value in the *reveal* phase. If there is a trapdoor known by some party, then the party can use it to efficiently retrieve (i.e., decrypt) the committed bit from any BitCom value. A practical instantiation is the Goldwasser-Micali probabilistic encryption scheme [GM84], assuming that modulo a Blum integer it is intractable for the *receiver* to decide quadratic residuosity (of residues with Jacobi Symbol 1). A bit 1 or 0 is committed by selecting a random group element and sending its square, or sending the additive inverse of its square, respectively.[7] To decommit 1 or 0, the *sender* reveals the bit and the respective random group element, letting the *receiver* verify that its square or additive-inverse of the square, respectively, is equal to the BitCom value. The factorization of the Blum integer is a trapdoor that enables efficient decision of quadratic residuosity.

Remark. The basis of the forge-and-lose technique (§4) is a combination of UB and UH BitCom schemes, with the *sender* in the UB scheme being the *receiver* in the UH scheme, and knowing a common trapdoor for both schemes. For the Blum integer examples, and assuming intractability of deciding quadratic residuosity (without a trapdoor), this would mean using the same Blum integer in both schemes, with its factorization as trapdoor. There are known protocols to prove correctness of a Blum integer (e.g., [vdGP88]).

The two exemplified schemes are XOR-homomorphic under modular multiplication. For the purpose of the new S2PC-with-BitComs protocol (§5), this homomorphism is useful in enabling efficient ZKPs *of knowledge* (ZKPoKs) related with committed bits, and efficient negotiation of random bit-encodings and respective BitComs (emulating an ideal functionality where the *trusted third party* would select the BitComs randomly). The property is also useful for linking several S2PC executions, via ZKPs about relations between the input bits of one execution and the input and output bits of previous executions (§6).

[7] The additive inverse of a square is necessarily a non-quadratic residue with Jacobi Symbol 1, modulo a Blum integer, because -1 has the same property.

3 The BitCom approach

This section introduces a BitCom approach that combines a BitCom setting (where there is a BitCom for each circuit input and output bit) and a C&C structure (where there are several GCs, each with two keys for each input and output wire). In this approach, based on the XOR-homomorphism of UH BitComs, the consistency of input and output wire keys across different GCs is *statistically* ensured within the C&C, rather than using a ZKP of consistency.[8]

3.1 Cut-and-choose stages

The S2PC-with-BitComs protocol to be defined in this paper is built on top of a C&C approach with a COMMIT-CHALLENGE-RESPOND-VERIFY-EVALUATE structure. In a COMMIT stage, P_A builds and sends several GCs, as well as complementary elements (dubbed *connectors*) related with BitComs and with the circuit input and output wire keys of GCs. At this stage, P_A does not yet reveal the circuit input keys that allow the evaluation of each GC. Then, in the CHALLENGE stage, P_A and P_B jointly decide a random partition of the set of GCs into two subsets, one for *verification* and the other for *evaluation*. Possibly, the subsets may be conditioned to a predefined restriction about their sizes (e.g., a fixed proportion of *verification* vs. *evaluation* GCs, or simply not letting the number of *evaluation* GCs exceed some value). In the subsequent RESPOND stage, P_A sends to P_B the elements that allow P_B to *fully verify* the correctness of the GCs selected for *verification*, to *partially verify* the connectors of all the GCs (in different ways, depending on whether they are associated with *verification* or *evaluation* challenges), and to *evaluate* the GCs (and respective connectors) selected for *evaluation*. In the VERIFY stage, if any verification step fails, then P_B aborts the protocol execution; otherwise, P_B establishes that there is an overwhelming probability that at least one GC (and respective connectors) selected for evaluation is correct. P_B finally proceeds to an EVALUATE stage, evaluating the *evaluation* GCs and respective connectors, and using their results to determine the final circuit output bits and respective decommitments of output BitComs. Notice that between the VERIFY and EVALUATE stages there is no *response* stage that could let P_A misbehave.

3.2 Connectors

This section develops the idea of *connectors* – structures used to sustain the integration between BitComs and the C&C structure. They are built on top of a setup where one initial UH-BitCom has been defined for each input and output wire of each party, independently of the number of GCs. Then, for each input and output wire in each GC, a connector is built to provide a (statistically verifiable) connection between the two BitCom decommitments and the respective pair of wire keys. The functionality of connectors varies with the type of wire they refer to (input of P_A, input of P_B, output of P_B), as illustrated in high level in Fig. 2.

Connectors are used in a type of commitment scheme (i.e., with *commit* and *reveal* phases) that takes advantage of the C&C substrate. First, each connector is committed in the C&C COMMIT stage, hiding the respective two wire keys, but binding P_A to them and to their relation with BitCom decommitments. Then, each connector is partially revealed

[8] The protocol still includes several efficient ZKPs related with BitComs, but they are not about the consistency of wire keys across different GCs.

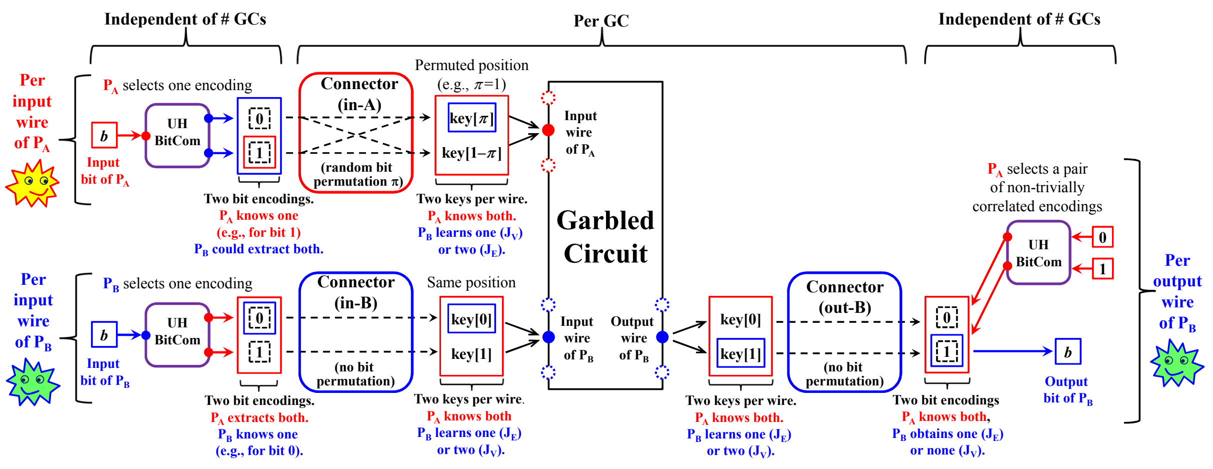

Independent of # GCs
Per GC
Independent of # GCs
Per input wire of P_A
P_A selects one encoding
b
Input bit of P_A
UH BitCom
0
1
Two bit encodings. P_A knows one (e.g., for bit 1) P_B could extract both.
Connector (in-A)
(random bit permutation π)
Permuted position (e.g., π=1)
key[π]
key[1−π]
Two keys per wire. P_A knows both. P_B learns one (J_V) or two (J_E).
Input wire of P_A
Garbled Circuit
Per input wire of P_B
P_B selects one encoding
b
Input bit of P_B
UH BitCom
0
1
Two bit encodings. P_A extracts both. P_B knows one (e.g., for bit 0).
Connector (in-B)
(no bit permutation)
Same position
key[0]
key[1]
Two keys per wire. P_A knows both P_B learns one (J_E) or two (J_V).
Input wire of P_B
Output wire of P_B
key[0]
key[1]
Two keys per wire. P_A knows both. P_B learns one (J_E) or two (J_V).
Connector (out-B)
(no bit permutation)
0
1
Two bit encodings P_A knows both, P_B obtains one (J_E) or none (J_V).
P_A selects a pair of non-trivially correlated encodings
UH BitCom
0
1
b
Output bit of P_B
Per output wire of P_B

Fig. 2: **Connectors.** Legend: $\mathrm{P_A}$ (GC *constructor*); $\mathrm{P_B}$ (GC *evaluator*); J_V and J_E (subsets of *verification* and *evaluation* GC indices, respectively); $\boxed{c}$ (group-element encoding bit c); key[c] (wire key with underlying bit c).

during the C&C RESPOND stage, in one of two possible complementary modes: a *reveal for verification*, related with *verification* GCs; or a *reveal for evaluation*, related with *evaluation* GCs. All verifications associated with these two reveal modes are performed in the C&C VERIFY stage, when $\mathrm{P_B}$ can still, immune to selective failure attacks, complain and abort in case it finds something wrong. $\mathrm{P_A}$ never executes simultaneously the two reveal modes for the same wire of the same GC, because such action would reveal the input bits (in case of wires of $\mathrm{P_A}$) or both BitCom decommitments (i.e., the trapdoor of $\mathrm{P_A}$, in case of wires of $\mathrm{P_B}$). Nonetheless, since the commitment to the *connector* binds $\mathrm{P_A}$ to the answers that it can give in each type of reveal phase, an incorrect connector can pass undetectably at most through one type of reveal mode. Thus, within the C&C approach, there is a negligible probability that $\mathrm{P_A}$ manages to build incorrect connectors for all evaluation indices and go by undetected. The specific constructions follow:

For each input wire of $\mathrm{P_A}$:

– **Commit.** $\mathrm{P_A}$ selects a random permutation bit and a respective random encoding (a group-element dubbed *multiplier*) using the same 2-to-1 square scheme used to commit the input bits of $\mathrm{P_A}$. $\mathrm{P_A}$ uses the homomorphic group operation to obtain a new encoding (dubbed *inner encoding*) that encodes the permuted version of her input bit, and sends its square (a new inner UH BitCom) to $\mathrm{P_B}$. $\mathrm{P_A}$ then builds a commitment of each of the two wire input keys (using some other commitment scheme), one for bit 0 and the other for bit 1, and sends them to $\mathrm{P_B}$ in the form of a pair with the respective permuted order.

– **Reveal for verification.** $\mathrm{P_A}$ decommits the two wire input keys (using the *reveal* phase of the respective commitment scheme), and decommits the permutation bit (by revealing the multiplier). $\mathrm{P_B}$ uses the two wire input keys (obtained for all input wires) to verify the correctness of the GC and simultaneously obtain the underlying bit of each input key. Then, $\mathrm{P_B}$ verifies that the ordering of the bits underlying the pair of revealed input keys is consistent with the decommitted permutation bit.

– **Reveal for evaluation.** $\mathrm{P_A}$ decommits the input key that corresponds to her input bit, and decommits the permuted input bit (by revealing the inner encoding), thus allowing $\mathrm{P_B}$ to verify that it is consistent with the position of the opened key commitment. As the value of the permuted bit is independent of the real input bit, nothing is revealed about the bit underlying the opened key. If $\mathrm{P_A}$ would instead reveal the other key, P_B would detect the cheating in a time when it is still safe to abort the execution and complain.

For each input wire of P_B:

- **Commit.** P_A selects a pair of random encodings of bit 0 (dubbed *multipliers*) and composes them homomorphically with the two known decommitments of the original input BitCom of P_B (which P_A has extracted using the trapdoor), thus obtaining two new independent encodings (dubbed *inner encodings*, one for bit 0 and one for bit 1). P_A then sends to P_B the respective squares (dubbed *inner squares*). For simplicity, it is assumed here that the inner encodings can be directly used as input wire keys of the GC (§B.2 shows how to relax this assumption).
- **Reveal for verification.** P_A reveals the two inner encodings. P_B verifies that they are the proper square-roots of the received inner squares, and that they encode bits 0 and 1, respectively. Then, P_A uses them as the circuit input keys in the GC verification procedure, verifying their correctness. A crucial point is that the two inner encodings are proper square-roots of independent BitComs and thus do not constitute a trapdoor.
- **Reveal for evaluation.** P_A reveals the two multipliers. P_B verifies that both encode bit 0, and homomorphically verifies that they are correct (their squares lead the original BitCom into the two received inner squares). Since P_B knows one (and only one) decommitment of the input BitCom, it can multiply it with the respective multiplier to learn the respective inner encoding and use it as an input wire key. This procedure is resilient to selective failure attack, because both multipliers are verified for correctness, and because the two inner encodings (of which P_B only learns one) are statistically correct input keys (i.e., they would be detected as incorrect if they had been associated with a *verification* GC).

For each output wire of P_B: The construction is essentially symmetric to the case of input wires of P_B. Again for simplicity, it is assumed here that the output keys can directly be group-elements (dubbed *inner encodings*) that are proper square-roots of independent squares. The underlying bit of each output key is thus the bit encoded by it (in the role of inner encoding). P_A commits by initially sending the two inner squares to P_B. Then, for *verification* challenges, from the GC verification procedure P_B learns 2 keys and respective underlying bits. P_B can verify that they are respective proper square-roots of the inner squares and that they encode the respective bits. For *evaluation* challenges, P_A sends only the two multipliers, and P_B verifies homomorphically that they are correct. Then, P_B learns one output key from the GC evaluation procedure, which is an inner encoding, and uses the respective multiplier to obtain the respective decommitment of the output BitCom.

The overall construction requires a number of group elements (multipliers and inner encodings) proportional to the number of input and output wires, but independent of the number of intermediate wires in the circuit.

4 The forge-and-lose technique

This section introduces a new technique, dubbed *forge-and-lose*, to improve the typical C&C-GCs-based approach, by using the BitCom approach to provide a new path for successful computation of final circuit output. More precisely, if in the EVALUATE stage there is at least one GC and respective connectors leading to a correct output (i.e., decommitments of the UH BitComs, for the correct circuit output bits), and if a malicious

$\mathrm{P}_{\mathrm{A}}^{*}$ successfully *forges* some other output, then $\mathrm{P}_{\mathrm{A}}^{*}$ *loses* the privacy of her input bits to P_{B}, allowing P_{B} to directly use a Boolean circuit to compute the intended output. This loss of privacy is not a violation of security, but rather a disincentive against malicious behavior by $\mathrm{P}_{\mathrm{A}}^{*}$.

The forge-and-lose path significantly reduces the probabilistic gap available for malicious behavior by P_{A} that might lead P_{B} to accept an incorrect output. The technique provides up to 1 bit of statistical security per GC, which constitutes an improvement factor of about 3.1 (either in reduction of number of GCs or in increase of number of bits of statistical security) in comparison with C&C-GCs that require a majority of correct *evaluation* GCs. As noted by Lindell [Lin13], in this setting the optimal C&C partition corresponds to an independent selection of verification and evaluation challenges. Still, for some efficiency tradeoffs it may be preferable to impose some restrictions on the number of *verification* and *evaluation* challenges (e.g., ensure that there are more *verification* than *evaluation* challenges). Appendix A shows the error probabilities associated with different C&C partition methods.

The forge-and-lose technique is illustrated in high level in Fig. 3. It can be merged into the C&C and BitCom approach as follows:

– **Encryption scheme.** P_{A} encrypts her own input bits using as key the trapdoor (known by P_{A}) of the UH-BitCom scheme used (by P_{A}) to produce BitComs of the output bits of P_{B}. Then, P_{A} gives a ZKP that her encrypted input is the same as that used in the S2PC protocol, i.e., the one committed by P_{A} with an UH-BitCom scheme with trapdoor known by P_{B}. If both schemes are XOR-homomorphic (see practical example in §2.2), the ZKP can be achieved efficiently with standard techniques, namely with a statistical combination across input wires, requiring communication linear with the statistical security parameter.

– **Forge-and-lose evaluation.** In the Evaluate stage, if a *connector* leads an output key to an invalid decommitment, then the respective GC is ignored altogether. If for the remaining GCs all connectors lead to consistent decommitments across all GCs, i.e., if for each output wire index the same valid bit-encoding (proper square-root of the output BitCom) is obtained, then P_{B} accepts them as correct. However, if P_{A} acted maliciously, there may be a forged GC and connector leading to a valid (verifiable) decommitment that is different from the decommitment obtained from another correct GC and connector, for the same output wire index. If P_{B} obtains any such pair of decommitments, i.e., a non-trivially correlated pair of square-roots of the same square, then P_{B} gets the trapdoor with which P_{A} encrypted her input, and follows to decrypt the input bits of P_{A} and use them directly to compute the correct final circuit output in the clear.

5 Protocol for 1-output S2PC-with-BitComs

This section describes the new C&C-GCs-based protocol for 1-output S2PC-with-BitComs, enhanced with a forge-and-lose technique. The BitComs are XOR-homomorphic, so the mentioned ZKPoKs are efficient using standard techniques.

0. Setup. The parties agree on the protocol goal, namely on a specification of a Boolean circuit whose evaluation result is to be learned privately by P_{B}, on the necessary security parameters, on a C&C partitioning method, and on the necessary sub-protocols.

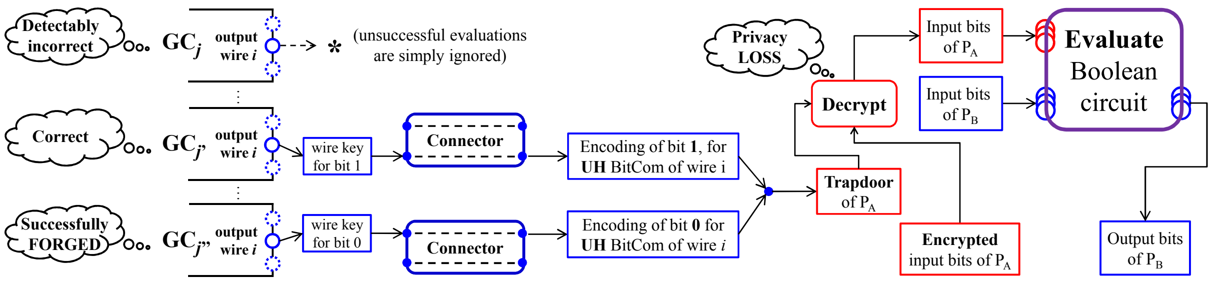


Fig. 3: **Forge-and-lose.** *Evaluation* path followed by P_B, the evaluator of *garbled circuits* (GCs), if different GCs built by a malicious P_A and selected for *evaluation* (e.g., with indices j', j'') lead to valid but different decommitments of the same *unconditionally hiding* (UH) BitCom (e.g., with index i).

Each party selects a *2-to-1 square* scheme, and proposes it to the other party, without revealing the trapdoor but giving a respective ZKPoK that proves the correctness of the public parameters.

1. Produce initial BitComs.
   (a) UH Commit Input Bits. Each party selects an initial UH BitCom for each of its own circuit input bits, using the 2-to-1 square scheme with trapdoor known by the other party, and sends it to the other party. P_B gives a ZKPoK of a valid decommitment of the respective BitComs.
   (b) UB Commit Input Bits of P_A. P_A commits again to each of her input bits, now using an UB-BitCom scheme with trapdoor equal to the trapdoor (known by P_A) of the UH-BitCom scheme used by P_B to commit the input bits of P_B. P_A gives a ZKPoK of equivalent decommitments between the UH BitComs of the input of P_A (with trapdoor known by P_B) and the UB BitComs of the input of P_A (with trapdoor known by P_A), i.e., a proof that the known decommitments encode the same bits.
   (c) UH Commit Output Bits of P_B. For each output wire index of P_B, P_A selects a random encoding of bit 0 (using the UH BitCom scheme with trapdoor known by P_A) and sends its square to P_B. (P_B will find a respective decommitment only later, in the Evaluate stage.)
2. Commit. P_A uses her trapdoor to extract a non-trivially correlated pair of proper square-roots from each UH BitCom of the input bits (this is the so called 2-out-of-1 OT, which replaces the typical 1-out-of-2 OT used in other S2PC protocols) and output bits of P_B. Then, P_A builds several GCs (in number consistent with the agreed parameters) and respective *connectors* to each input and output wire, and sends the GCs and commitments to the connectors (as specified in §3.2) to P_B.
3. Challenge. The two parties use a coin-tossing sub-protocol to determine a random challenge bit for each GC, conditioned to the agreed C&C method (e.g., same number of challenges of each type, or more verification than evaluation challenges, or independent selection).[9]
4. Decide UH-BitCom Permutations. In order to emulate a *trusted third party* deciding the UH BitCom of each circuit input and output bit, both parties interact in a fully-simulatable coin-tossing sub-protocol to decide a random encoding of bit 0

[9] The standalone coin-tossing (see an instantiation in Fig. 7, §C) does not need to be fully simulatable, but the proof of security takes advantage of the ability of the simulated P_A (with rewinding access to a possibly malicious P_B^*) to decide the outcome of the coin-toss. Subtle alternatives would be possible, depending on some changes related with the remaining stages.

for each wire index.[10] Later, each party will locally use these encodings to permute the encodings of her respective private bits, and use the square of the encodings to permute the respective UH BitComs of both parties. Given the XOR-homomorphism, the initial and the final UH BitComs commit to the same bits.

5. RESPOND. For each C&C challenge bit, $\mathrm{P_A}$ makes either the *reveal for verification* or the *reveal for evaluation* of the connectors, as specified in §3.2.
6. VERIFY. For *verification* indices, $\mathrm{P_B}$ obtains two keys per input wire, verifies the correctness of the GC and makes the respective partial verification of connectors (without learning the decommitments of the BitComs of output bits of $\mathrm{P_B}$). For *evaluation* indices, $\mathrm{P_B}$ makes the respective partial verification of the connectors and obtains one key per input wire. If something is found wrong, $\mathrm{P_B}$ aborts and outputs FAIL.
7. EVALUATE. For each *evaluation* index, $\mathrm{P_B}$ uses the one key per input wire to evaluate the GC, obtain one key per output wire and use the respective revealed part of the connector (namely, one of the two received multipliers) to obtain a decommitment (bit encoding) of the respective output BitCom. There is an overwhelming probability that there is at least one *evaluation* GC whose connectors lead to valid decommitments in all output wires. If all obtained valid decommitments are consistent across different GCs, then $\mathrm{P_B}$ accepts them as correct. Otherwise, $\mathrm{P_B}$ proceeds into the forge-and-lose path as follows. It finds a non-trivially correlated pair of square-roots and uses it as a trapdoor to decrypt the input bits of $\mathrm{P_A}$, from the respective UB BitComs. In possession of the input bits of both parties, $\mathrm{P_B}$ directly evaluates the final circuit output. Then, from within the decommitments already obtained from the *evaluation* connectors, $\mathrm{P_B}$ finds the output bit encodings that are consistent with the circuit output bits, and accepts them as the correct ones. This marks the end of the forge-and-lose path.
8. APPLY BITCOM PERMUTATIONS. Each party applies the previously decided random permutations to the encodings of the respective circuit input and output bits, and applies the square of the random encodings as permutations to the UH BitComs of the circuit input and output bits of both parties.
9. FINAL OUTPUT. Each party privately outputs her circuit input and output bits and the respective final encodings, and also outputs the (commonly known) final UH BitComs of the circuit input and output bits of both parties. $\mathrm{P_A}$ outputs even if $\mathrm{P_B}$ aborts at any time after the APPLY BITCOM PERMUTATIONS stage.

Remark. When using the 1-output protocol within larger protocols, care needs to be taken so that $\mathrm{P_A}$ cannot distinguish between $\mathrm{P_B}$ having learned his output via the normal evaluation path vs. via the forge-and-lose path.

6 Discussion

6.1 Complexity

Besides the computation and communication related with (the reduced number of) GCs, the new S2PC-with-BitComs protocol requires instantiating the connectors (which brings a cost proportional to the number of input and output wires, multiplied by the number of

[10] To achieve simulability of the overall protocol under each possible malicious party ($\mathrm{P_A^*}$ and $\mathrm{P_B^*}$), the simulator of this coin-tossing needs to be able to induce the final BitComs in the real world to be equal to those decided by the trusted third party in the ideal world, and at the same time deal with a probabilistic possibility of abort dependent on those final BitCom values (e.g., see [Lin03]).

GCs), performing ZKPoKs related with BitComs and to prove correctness of the BitCom scheme parameters, and performing secure two-party coin-tossing (which is significant for the decision of random BitComs values). Based on the XOR-homomorphism, the ZKPoKs related with input wires can be parallelized efficiently with standard techniques, with a communication cost linear in a statistical parameter but independent of the number of input wires, though with computational cost proportional to the product of the statistical parameter and the number of input wires.

With an instantiation based on Blum integers, the inversion of an UH BitCom using the trapdoor (i.e., computing a modular square-root) is approximately computationally equivalent to one exponentiation modulo each prime factor. Thus, besides proving correctness of the Blum integer (which can be achieved with a number of exponentiations that is linear in the statistical parameter), and performing a fully-simulatable coin-tossing sub-protocol to decide random BitCom permutations (which can be instantiated with a number of exponentiations that is linear in the number of input and output wires, and performed in a group of smaller order), the 1-output S2PC-with-BitComs protocol only requires a number of exponentiations that is linear in the number of input wires of $\mathrm{P_B}$, and only computed by $\mathrm{P_A}$. This is in contrast with other protocols whose required number of exponentiations by both parties is proportional to the number of GCs multiplied by the number of input wires (e.g., [LP11]), though in compensation those exponentiations are supported in groups with smaller moduli length and sub-groups of smaller order.

The protocol can be optimized in several ways (see §E). For example, with a *random seed checking* (RSC) technique [GMS08] the communication of elements (including GCs and connectors) associated with verification challenges can be replaced by the sending and verification of small random seeds (used to pseudo-randomly generate the elements) and a commitment (to the elements). The technique can be applied independently to GCs and connectors, and can also be used to reduce some of the communication corresponding to connectors associated with *evaluation* challenges. As another example, some group elements used in connectors of $\mathrm{P_A}$ can be reduced in size, since their binding properties only need to hold during the execution of the protocol.

Concrete results. An analytic estimation of communication complexity is made in §E (ignoring overheads due to communication protocols), for two different circuits: an AES-128 circuit with 6,800 multiplicative gates [Bri13] and 128 wires for the input of each party and for the output of $\mathrm{P_B}$; and a SHA-256 circuit with 90,825 multiplicative gates [Bri13] and 256 wires for the input of each party and output of $\mathrm{P_B}$.

An interesting metric is the proportional overhead of communicated elements beyond GCs (i.e., connectors, BitComs and associated proofs) in comparison with the size occupied only by the GCs. For 128 bits of cryptographic security, instantiated with 3,072-bit Blum integers [BBB+12], and 40 bits of statistical security achieved using 41 GCs of which at most 20 are for evaluation (see §A), the estimated overhead is about 55% and 8%, for the AES-128 and SHA-256 circuits, respectively, without the RSC technique applied to the GCs. This metric gives an intuition about the communication cost inherent to the BitCom approach, but is not good enough on its own. For example, when applying the RSC technique also at the level of GCs, the overall communication is reduced significantly, but (because the size corresponding to GCs is reduced) the proportional overhead increases to 158% and 23%, respectively. Nonetheless, even these overheads are low when compared to the cost associated with the additional GCs needed in a C&C that requires a majority of correct evaluation GCs (i.e., on its own an overhead of about 200%, and asymptotically up to about 210%). Clearly, the proportional overhead decreases with the ratio given by the number of input and output wires divided the number of multiplicative gates.

There are other optimizations and C&C configurations that reduce the communication even more, with tradeoffs with computational complexity. For example, by restricting the number of *evaluation* GCs to be at most 8, but increasing the overall number of GCs to 123 (this was the minimal number of GCs required by the typical C&C to achieve 40 bits of statistical security), the estimated communication complexity is approximately of the order of 62 million bits and 418 million bits, respectively for the exemplified circuits. A *pipelining* technique [HEKM11] could also be considered, such that the garbled-gates are not all stored in memory at the same time. This would increase the computation by $\mathrm{P_A}$, but not affect the amount of communicated elements.

6.2 Linked executions

A simple example of linked executions is the mentioned dual-path execution approach, where each party reuses the same input bits (and BitComs) in two different executions. Furthermore, it may be useful to achieve more general linkage, such as proving that the private input bits of a S2PC satisfy certain *non-deterministic polynomial* verifiable relations with the private input and output bits of previous S2PCs. Based on the XOR-homomorphism of BitComs, this can be proven with efficient ZKPs. For example, proving that a certain BitCom commits to the NAND of the bits committed by two other BitComs can be reduced to a simple ZKP that there are at least two 1's committed in a triplet of BitComs, with the triplet being built from a XOR-homomorphic combination of the original three BitComs.[11]

For example, since Boolean circuits can be implemented with NAND gates alone, it is possible to prove, outside of the GCs, those transformations and relations that involve only the bits of one party. For example, for protocols defined as a recursion of small GC-based S2PC sub-protocols in the semi-honest model (e.g., [LP02]), security can be enhanced to resist also the malicious model, by simply (1) replacing each GC with a C&C-GCs with BitComs, and (2) by naturally using the input and output of previous executions (or transformations thereof) as the input of the subsequent executions.

6.3 Security

The protocol can be proven secure in the plain model (i.e., without hybrid access to ideal functionalities), assuming the simulator has black-box access with rewindable capability to a real adversary (§G). The simulator is able to extract the input of the malicious party in the real world from the respective ZKPoKs of decommitments, and thus hand it over to the *trusted third party* in the ideal world. The two-party coin tossing used to select random permutations of group-elements needs to be fully-simulatable, because the final BitComs and decommitments are also part of the final output of honest parties. Subtle changes are needed to the ideal functionality when the protocol is adjusted to the 2-output case where each party learns a private circuit output. Achieving security in the *universal composability* model [CLOS02] is left for future work.

[11] The first bit is the NAND of the two last if and only if there are at least two 1's in the triplet composed of the first bit and of the XOR of the first bit with each of the other two bits [Bra06]. See details in Fig. 16 in appendix.A different method can be found in [BDP00].

7 Related work

The paragraphs below compare some aspects of related work.

7.1 Two other optimal C&C-GCs

Two recently proposed C&C-GCs-based protocols [Lin13; HKE13] also minimize the number of GCs, requiring only that at least one evaluation GC is correct.

Lindell [Lin13] enhances a typical C&C-GCs-based protocol by introducing a second C&C-GCs, dubbed *secure-evaluation-of-cheating* (SEOC), where $\mathrm{P_B}$ recovers the input of $\mathrm{P_A}$ in case $\mathrm{P_B}$ can provide two different garbled output values from the first C&C-GCs. The concept of input-recovery resembles the forge-and-lose technique, but the methods are quite different. For example, the SEOC phase requires interaction between the parties after the first GC evaluation phase, whereas in the forge-and-lose the input-recovery occurs offline.

Huang, Katz and Evans [HKE13] propose a method that combines the C&C-GCs approach with a verifiable secret sharing scheme (VSSS). The parties play different roles in two symmetric C&C-GCs, and then securely compare their outputs. This requires the double of GCs, but in parallel across the two parties. By requiring a predetermined number of verification challenges, the necessary number of GCs is only logarithmically higher than the optimal that is achieved with an independent selection of challenges. In their method, the deterrent against optimal malicious GCs construction does not involve the GC constructor party having her input revealed to the GC evaluator.

In the SEOC and VSSS descriptions, the method of ensuring input consistency across different GCs is supported on discrete-log based intractability assumptions. The descriptions do not consider general linkage of S2PC executions related with output bits, but their input bits are also committed using XOR-homomorphic BitComs. In contrast, the S2PC-with-BitComs described in this paper, with an instantiation based on Blum integers, is based on intractability of deciding quadratic residuosity and requires a lower number of exponentiations, though with each exponentiation being more expensive due to the larger size of group elements and group order, for the same cryptographic security parameter. Future work may better clarify the tradeoffs between the three techniques.

7.2 Other related work

Jarecki and Shmatikov [JS07] described a S2PC protocol with committed inputs, using a single verifiably-correct GC, but with the required number of exponentiations being linear in the number of gates. In comparison, the protocol in this paper allows garbling schemes to be based on symmetric primitives (e.g., block-ciphers, whose greater efficiency over-compensates the cost of multiple GCs in the C&C), and the required number of exponentiations to be linear in the number of circuit input and output bits and in the statistical parameter.

Nielsen and Orlandi proposed LEGO [NO09], and more recently Frederiksen et al. proposed Mini-Lego [FJN^{+}13], a fault-tolerant circuit design that computes correctly even if some garbled gates are incorrect. Their protocol, which uses a cut-and-choose at the garbled-gate level (instead of at the GC level) to ensure that most garbled gates used for evaluation are correct, requires a single GC but of larger dimension. It would be interesting to explore, in future work, how to integrate a forge-and-lose technique into their cut-and-chose at the gate level.

Kolesnikov and Kumaresan [KK12] described a S2PC slice-evaluation protocol, based on information theoretic GCs, allowing the input of one GC to directly use the output of a previous GC. Their improvements are valid if the linked GCs are shallow, and if one party is semi-honest and the other is covert. In contrast, the S2PC-with-BitComs protocol in this paper allows any circuit depth and any party being malicious.

Nielsen et al. [NNOB12] proposed an OT-based approach for S2PC, potentially more efficient than a C&C-GCs if network latency is not an issue. However, the number of communication rounds of their protocol is linear in the depth of the circuit, thus being outside of the scope of this paper (restricted to C&C-GCs-based protocols with a constant number of communication rounds).

Acknowledgments. The author thanks: his Ph.D. co-advisor Alysson Bessani, for the valuable discussions and suggestions that contributed to improve the presentation of an earlier version of this paper; another reviewer, who wished to remain anonymous, for valuable suggestions that also contributed to improve the presentation of an earlier version of this paper; the anonymous reviewers of CRYPTO 2013 and ASIACRYPT 2013 conferences, for their useful comments.

References

[BBB$^+$12] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for Key Management – Part 1: General (Revision 3) – NIST Special Publication 800-57. U.S. Department of Commerce, NIST-ITL-CSD, July 2012. 15, 47

[BCC88] G. Brassard, D. Chaum, and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, 1988. 7

[BDP00] J. Boyar, I. Damgård, and R. Peralta. Short Non-Interactive Cryptographic Proofs. *J. Cryptology*, 13:449–472, 2000. 16

[Bea96] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proc. STOC '96*, pages 479–488. ACM, New York, 1996. 6

[BHR12] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proc. CCS '12*, pages 784–796. ACM, New York, 2012. See also Cryptology ePrint Archive, Report 2012/265. 4, 5

[Blu83] M. Blum. Coin flipping by telephone a protocol for solving impossible problems. *SIGACT News*, 15:23–27, January 1983. 7

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proc. STOC '90*, pages 503–513. ACM, New York, 1990. 5

[Bra06] L. T. A. N. Brandão. A Framework for Interactive Argument Systems using Quasigroupic Homorphic Commitment. Cryptology ePrint Archive, Report 2006/472, 2006. 16

[Bri13] Bristol Cryptography Group. Circuits of Basic Functions Suitable For MPC and FHE. `http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`, Accessed June 2013. 15, 46

[Can00] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology*, 13:143–202, 2000. See also Cryptology ePrint Archive, Report 1998/018. 3, 61

[CGT95] C. Crépeau, J. v. d. Graaf, and A. Tapp. Committed Oblivious Transfer and Private Multi-Party Computation. In D. Coppersmith, editor, *CRYPTO '95*, vol. 963 of *LNCS*, pages 110–123. Springer-Verlag, 1995. 6

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *Proc. STOC '02*, pages 494–503. ACM, New York, 2002. See also Cryptology ePrint Archive, Report 2002/140. 16

[CP93] D. Chaum and T. Pedersen. Wallet Databases with Observers. In E. Brickell, editor, *CRYPTO '92*, vol. 740 of *LNCS*, pages 89–105. Springer-Verlag, 1993. 38

[Dam88] I. B. Damgård. *The application of claw free functions in cryptography*. PhD thesis, Aarhus University, Mathematical Institute, 1988. 7

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28:637–647, June 1985. 6

[ElG85] T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In G. Blakley and D. Chaum, editors, *Advances in Cryptology*, vol. 196 of *LNCS*, pages 10–18. Springer-Verlag, 1985. 38

[FFS88] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *J. Cryptology*, 1(2):77–94, 1988. 52

[FJN+13] T. Frederiksen, T. Jakobsen, J. Nielsen, P. Nordholt, and C. Orlandi. MiniLEGO: Efficient Secure Two-Party Computation from General Assumptions. In T. Johansson and P. Nguyen, editors, *EUROCRYPT '13*, vol. 7881 of *LNCS*, pages 537–556. Springer-Verlag, 2013. See also Cryptology ePrint Archive, Report 2013/155. 17

[FN13] T. Frederiksen and J. B. Nielsen. Fast and Maliciously Secure Two-Party Computation Using the GPU. In M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *ACNS '13*, vol. 7954 of *LNCS*, pages 339–356. Springer-Verlag, 2013. 3

[GK96] O. Goldreich and A. Kahan. How to Construct Constant-Round Zero-Knowledge Proof Systems for NP. *J. Cryptology*, 9(3):167–189, 1996. 63

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984. 8

[GMPY06] J. Garay, P. MacKenzie, M. Prabhakaran, and K. Yang. Resource Fairness and Composability of Cryptographic Protocols. In *TCC '06*, vol. 3876 of *LNCS*, pages 404–428. Springer-Verlag, 2006. See also Cryptology ePrint Archive, Report 2005/370. 69

[GMR84] S. Goldwasser, S. Micali, and R. L. Rivest. A "Paradoxical" Solution To The Signature Problem. In *Proc. FOCS '84*, pages 441–448. IEEE Computer Society, 1984. 7

[GMS08] V. Goyal, P. Mohassel, and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In N. Smart, editor, *EUROCRYPT '08*, vol. 4965 of *LNCS*, pages 289–306. Springer-Verlag, 2008. 4, 15, 24, 40

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proc. STOC '87*, pages 218–229. ACM, New York, 1987. 5

[Gol04] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, chapter 7 edition, 2004. 2

[HEKM11] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proc. SEC '11*. USENIX Association, 2011. 4, 16, 49

[HKE12] Y. Huang, J. Katz, and D. Evans. Quid-Pro-Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution. In *Proc. S&P '12*, may 2012. 7

[HKE13] Y. Huang, J. Katz, and D. Evans. Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose. In R. Canetti and J. Garay, editors, *CRYPTO '13*, vol. 8043 of *LNCS*, pages 18–35. Springer-Verlag, 2013. See also Cryptology ePrint Archive, Report 2013/081. 4, 7, 17

[HL10] C. Hazay and Y. Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *J. Cryptology*, 23(3):422–456, 2010. 38

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In D. Boneh, editor, *CRYPTO '03*, vol. 2729 of *LNCS*, pages 145–161. Springer-Verlag, 2003. 6

[JS07] S. Jarecki and V. Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. In M. Naor, editor, *EUROCRYPT '07*, vol. 4515 of *LNCS*, pages 97–114. Springer-Verlag, 2007. 17

[KAF+10] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik, H. Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-Bit RSA Modulus. In T. Rabin, editor, *CRYPTO '10*, vol. 6223 of *LNCS*, pages 333–350. Springer-Verlag, 2010. See also Cryptology ePrint Archive, Report 2010/006. 42

[Kir08] M. S. Kiraz. *Secure and Fair Two-Party Computation*. Phd thesis, Technische Universiteit Eindhoven, Netherlands, 2008 2008. 7

[KK12] V. Kolesnikov and R. Kumaresan. Improved Secure Two-Party Computation via Information-Theoretic Garbled Circuits. In I. Visconti and R. De Prisco, editors, *SCN '12*, vol. 7485 of *LNCS*, pages 205–221. Springer-Verlag, 2012. 6, 18

[Kol09] V. Kolesnikov. Advances and impact of secure function evaluation. *Bell Labs Technical Journal*, 14(3):187–192, 2009. 2

[KS06] M. S. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of Yao's garbled circuit construction. In *Proc. 27th Symp. Information Theory in the Benelux*, pages 283–290, 2006. 6, 70

[KS08a] M. S. Kiraz and B. Schoenmakers. An Efficient Protocol for Fair Secure Two-Party Computation. In T. Malkin, editor, *CT-RSA '08*, vol. 4964 of *LNCS*, pages 88–105. Springer-Verlag, 2008. 6

[KS08b] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP '08*, vol. 5126 of *LNCS*, pages 486–498. Springer-Verlag, 2008. 4

[KSS12] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proc. Security '12*, pages 285–300. USENIX Association, 2012. See also Cryptology ePrint Archive, Report 2012/179. 3, 42, 49

[Lin03] Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. *J. Cryptology*, 16(3):143–184, 2003. See also Cryptology ePrint Archive, Report 2001/107. 14, 38, 39

[Lin13] Y. Lindell. Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries. In R. Canetti and J. Garay, editors, *CRYPTO '13*, vol. 8043 of *LNCS*, pages 1–17. Springer-Verlag, 2013. See also Cryptology ePrint Archive, Report 2013/079. 4, 12, 17, 23, 48

[LP02] Y. Lindell and B. Pinkas. Privacy Preserving Data Mining. *J. Cryptology*, 15(3):177–206, 2002. 2, 16

[LP07] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In M. Naor, editor, *EUROCRYPT '07*, vol. 4515 of *LNCS*, pages 52–78. Springer-Verlag, 2007. See also Cryptology ePrint Archive, Report 2008/049. 6

[LP09] Y. Lindell and B. Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptology*, 22(2):161–188, 2009. 5, 67

[LP11] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Y. Ishai, editor, *TCC '11*, vol. 6597 of *LNCS*, pages 329–346. Springer-Verlag, 2011. See also Cryptology ePrint Archive, Report 2010/284. 6, 15, 39, 40

[LPS08] Y. Lindell, B. Pinkas, and N. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In R. Ostrovsky, R. De Prisco, and I. Visconti, editors, *SCN '08*, vol. 5229 of *LNCS*, pages 2–20. Springer-Verlag, 2008. 38, 39

[MF06] P. Mohassel and M. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC '06*, vol. 3958 of *LNCS*, pages 458–473. Springer-Verlag, 2006. 6, 7, 70

[NNOB12] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO '12*, vol. 7417 of *LNCS*, pages 681–700. Springer-Verlag, 2012. See also Cryptology ePrint Archive, Report 2011/091. 6, 18

[NO09] J. B. Nielsen and C. Orlandi. LEGO for Two-Party Secure Computation. In O. Reingold, editor, *TCC '09*, vol. 5444 of *LNCS*, pages 368–386. Springer-Verlag, 2009. See also Cryptology ePrint Archive, Report 2008/427. 6, 17

[NP01] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA '01*, pages 448–457. SIAM, Philadelphia, 2001. 6

[NPS99] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proc. EC '99*, pages 129–139. ACM, New York, 1999. 4, 5

[NZM91] I. M. Niven, H. S. Zuckerman, and H. L. Montgomery. *An introduction to the theory of numbers*. Wiley, fifth edition, 1991. 7, 56

[Pin03] B. Pinkas. Fair Secure Two-Party Computation. In E. Biham, editor, *EUROCRYPT '03*, vol. 2656 of *LNCS*, pages 647–647. Springer-Verlag, 2003. 6

[PSSW09] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-Party Computation Is Practical. In M. Matsui, editor, *ASIACRYPT '09*, vol. 5912 of *LNCS*, pages 250–267. Springer-Verlag, 2009. See also Cryptology ePrint Archive, Report 2009/314. 2, 4, 6, 47

[Rab81] M. O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Harvard University, Aiken Computation Lab, Cambridge, MA, 1981. See typesetted version in Cryptology ePrint Archive, Report 2005/187. 6

[Sch91] C. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991. 38

[SS11] A. Shelat and C.-h. Shen. Two-Output Secure Computation with Malicious Adversaries. In K. Paterson, editor, *EUROCRYPT '11*, vol. 6632 of *LNCS*, pages 386–405. Springer-Verlag, 2011. See also Cryptology ePrint Archive, Report 2011/533. 4, 6, 7, 22

[TLL03] C. Tang, Z. Liu, and J. Liu. The Statistical Zero-knowledge Proof for Blum Integer Based on Discrete Logarithm. Cryptology ePrint Archive, Report 2003/232, 2003. 60

[vdGP88] J. van de Graaf and R. Peralta. A Simple and Secure Way to Show the Validity of Your Public Key. In C. Pomerance, editor, *CRYPTO '87*, vol. 293 of *LNCS*, pages 128–134. Springer-Verlag, 1988. 8, 58

[Woo07] D. P. Woodruff. Revisiting the Efficiency of Malicious Two-Party Computation. In M. Naor, editor, *EUROCRYPT '07*, vol. 4515 of *LNCS*, pages 79–96. Springer-Verlag, 2007. See also Cryptology ePrint Archive, Report 2006/397. 6

[Yao82] A. C. Yao. Protocols for secure computations. In *Proc. FOCS '82*, pages 160–164. IEEE Computer Society, 1982. 2

[Yao86] A. C.-C. Yao. How to generate and exchange secrets. *FOCS '86*, 0:162–167, 1986. 5

Appendix

This appendix includes formal details related with claims mentioned along the main text. Section A compares the error probability related with a C&C-with-forge-and-lose vs. that of a C&C requiring a majority of correct evaluation GCs, and analyzes different C&C configurations. Section B specifies the *connectors* that sustain the BitCom approach, based on XOR-homomorphic properties. Section C specifies with better detail the 1-output S2PC-with-BitComs protocol. Section D looks into a fully-simulatable coin-tossing subprotocol that can be used to obtain random BitCom permutations. Section E discusses several optimizations and estimates the respective communication complexity of the protocol. Section F describes how to accomplish the ZKPoKs related with 2-to-1 square schemes. Section G gives the proof of security. Section H provides an index of notation. The last page also of this document contains the list of Figures and the list of Tables.

A Soundness error probability

This section relates the number of GCs with the number of bits of statistical security of different C&C partition methods. By definition, the number of bits of statistical security is the additive inverse of the logarithm base 2 of the maximum error probability, i.e., for which a malicious PA can make PB accept an incorrect output. It is assumed that a malicious P_A^* uses the optimal strategy to lead P_B to accept an incorrect output. The calculated probabilities are valid for C&C protocols where a *bad* index (i.e., one in which P_A has cheated in the COMMIT stage) is detected if selected for *verification*, as is the case of the protocol defined in this paper. Table 1 shows values or error probabilities; Table 2 shows the number of GCs necessary to achieve certain values of statistical security.

For soundness to be broken, all the committed elements associated with the v indices selected for *verification* must be *good* (i.e., correctly constructed) – otherwise, P_B safely aborts the execution upon detection of a *bad* element. Furthermore, within the remaining e indices selected for *evaluation*, the number b of *bad* indices must be enough to lead P_B to an incorrect result. The optimal strategy of a malicious P_A^* to succeed in breaking soundness depends on the C&C partitioning method (e.g., fixed vs. variable number of verification indices) and the requirements for soundness (e.g., the required number of *good* evaluation indices: a majority vs. at least one).

The formula for error probability follows from a simple counting argument. The total number of possible C&C choices is equal to the number of possible sets of verification indices that can be selected from the set of all indices. Of these, the choices that lead to error are those for which all the *bad* elements are not selected for verification (and assuming that there are enough bad indices to induce error). The number of these choices is equal to the number of possible sets of verification indices that can be selected from the set of *good* indices. The error probability is given by the quotient of the later quantity over the former quantity. Table 1 shows the error probabilities associated with different C&C methods, namely methods where the number of *verification* and *evaluation* indices are pre-determined, and methods where the number of challenges of each type is variable.

Methods requiring a majority of correct evaluation indices. In C&C methods where the final output is determined as the majority output of the e evaluated GCs, a fixed partition size is preferable to a variable one, because it yields a lower probability of error (1). In these cases, the optimal adversarial approach by P_A (the GC constructor)

Table 1: Soundness error probability

Correctness requirement	C&C partition method		b (# bad indices)	Error probability (Pr)		Bits of statistical security ($-\log_2$Pr) (approximate)	Eq. #
	Mode	Restriction		Exact	Stirling's approximation		
A majority of evaluation indices is good	Fixed	$\langle v, e\rangle$	$\lceil e/2\rceil$ (fixed)	$\frac{\text{Bin}(s-b,v)}{\text{Bin}(s,v)} = \frac{(s-b)!e!}{(e-b)!s!}$	...	...	(1)
		$v \approx s/2$		$\approx \frac{(3s/4)!(s/2)!}{(s/4)!s!}$	$2^{-(1+3s)/2} \cdot 3^{(2+3s)/4}$	$0.311s - 0.292$	(2)
		$v \approx 3s/5$		$\approx \frac{(4s/5)!(2s/5)!}{(s/5)!s!}$	$2^{3/2} \cdot 5^{-1/2} \cdot (5/4)^{-s}$	$0.322s - 0.339$	(3)
At least one evaluation index is good	− − −	$\langle v, e\rangle \approx \langle \alpha s, \beta s\rangle$	e (fixed)	$e!v!/s!$	$\sqrt{2\pi\alpha\beta s}(\alpha^\alpha \beta^\beta)^s$	$c_1 s - (\log_2 s)/2 - c_0$	(4)
		$v = \lceil s/2\rceil$		$\lfloor s/2\rfloor! \lceil s/2\rceil!/s!$	$2^{-s}\sqrt{\pi \cdot s/2}$	$s - (\log_2 s)/2 - 0.326$	(5)
	Variable	$0 < e \le e'$: Pr $\lessapprox 2^{-4e'}$	e (variable)	...	...	$\gtrapprox 4e'$	(6)
		$0 < e \le v < s$		$\le 1/(2^{s-1} - 1) \approx 2^{-(s-1)}$		$\gtrapprox s - 1$	(7)
		$0 \le v < s$		$1/(2^s - 1) \approx 2^{-s}$		$\approx s$	(8)

Legend: # (number of), s (# challenge indices), v (# indices selected for verification), e (# indices selected for evaluation), b (# bad indices); e' (auxiliary parameter defining a statistical security goal). By definition, $s = v+e$ and $\alpha+\beta = 1$, with $\alpha, \beta > 0$. Bin$(\cdot, \cdot)$ denotes the binomial coefficient, with Bin$(n, m) = n!/(m!(n-m)!)$. Stirling's first order approximation establishes $n! \approx \sqrt{2\pi n}(n/E)^n$, as n approaches infinite, where $E \approx 2.7828$ is Euler's constant (the basis of the natural logarithm), and $\pi \approx 3.1416$ is the quotient between the circumference and the diameter of a circle ☺. $c_1 = \log_2(\alpha^{-\alpha}\beta^{-\beta})$; $c_0 = (\log_2(2\pi\alpha\beta))/2$. **Remark:** In (5), using $v = \lfloor s/2\rfloor$ would lead to the same error probability as using $v = \lceil s/2\rceil$. However, in (2), when s is odd there is a difference in probability when comparing the case $v = \lceil s/2\rceil$ with the case $v = \lfloor s/2\rfloor$, depending on the value $s(\text{mod } 4)$. When $s(\text{mod } 4) = 3$ then $v = \lceil s/2\rceil$ is better by 1 bit of statistical security, i.e., it yields half the probability. When $s(\text{mod } 4) = 1$ then $v = \lfloor s/2\rfloor$ can be better up to about 0.5 bits of statistical security.

is to forge elements in the *minimal number* of indices that might prevent a *majority* of evaluation indices from being correct. In other words, the number of *bad* indices must be the lowest number that constitutes at least half of the number of evaluation indices, thus minimizing the probability of being detected in the VERIFY stage. A common C&C partition method is to split the challenges in about half for verification and half for evaluation, which yields an error probability corresponding to about 0.31 bits of statistical security per GC (2). However, the optimal proportion is achieved [SS11] with about three fifths of indices selected for verification and two fifths for evaluation (3), improving the statistical security up to about 0.32 bits per GC.

Methods requiring at least one correct evaluation index. With the forge-and-lose technique, a successful attack by P_A requires cheating in exactly as many indices as those that will be selected for evaluation. Furthermore, soundness is broken only if P_A guesses the exact partition configuration in advance. The error probability obtained when selecting fixed proportions of challenge types (4) is better (i.e., lower) than when using the same proportions in a technique that requires a majority of *good* evaluation indices (compare with (1)). In both cases the *soundness error* probability is negligible in s, but the methods that just require at least one *good* evaluation index are clearly better. The optimal fixed proportion is half of indices for evaluation and the remaining half for verification, because this is the number that maximizes the number of possible partitions (out of which a malicious P_A^* has to guess one) (5). The respective probability corresponds to about 1 bit of statistical security per GC, except for a reduction by an (additive) logarithmic factor. Table 2 shows the minimum numbers of GCs that achieve certain values of statistical security. For example, to achieve 128 bits of security, the forge-

Table 2: Number of GCs to achieve statistical security

Correctness requirement	C&C partition method		Probability goal	2^{-40}	2^{-80}	2^{-128}
	More	Restriction				
Majority of evaluation indices are good	Fixed	$v = \lfloor s/2 \rfloor$ (half-half)	(s, v, e)	(129, 64, 65)	(257, 128, 129)	(410, 205, 205)
			Bits security	40.54	80.39	128.12
		$(v \approx 3s/5)$ (fixed optimal)	(s, v, e)	(123, 74, 49)	(247, 150, 97)	(396, 239, 157)
			Bits security	40.26	80.17	128.15
At least one evaluation index is good		$(v = \lceil s/2 \rceil)$ (half-half) (fixed optimal)	(s, v, e)	(44, 22, 22)	(84, 42, 42)	(132, 66, 66)
			Bits security	40.94	80.47	128.15
	Variable	$(e \leq e' : \Pr \approx 2^{-4e'})$ (low e)	(s, v, e)	(76, 66, 10)	(142, 122, 20)	(220, 188, 32)
			Bits security	40.02	80.15	128.19
		$\lceil s/2 \rceil \leq v < s$ (more v than e)	(s, v, e)	(41, ?, ?)	(81, ?, ?)	(129, ?, ?)
			Bits security	40.32	80.23	128.19
		$v < s$ (fully independent)	(s, v, e)	(40, ?, ?)	(80, ?, ?)	(128, ?, ?)
			Bits security	40	80	128

Legend: # (number of), s (# challenge indices), v (# indices selected for verification), e (# indices selected for evaluation), ? (a variable number between 1 and s).

and-lose technique with the optimal fixed proportions (half-half) only requires 132 GCs, instead of the 396 required by typical C&Cs-GC techniques with the respective optimal fixed proportions (3-fifths-2-fifths).

Independent selection of challenges. For a fixed number of GCs, the error probability can be slightly improved by not pre-determining the number of evaluation challenges. In particular, by allowing any possible C&C configuration, the probability that P_A guesses the exact configuration corresponds to exactly 1 bit of statistical security per GC (as noticed in [Lin13]). As a result, for the same error probability threshold there is a logarithmic (additive) reduction in number of GCs (compare (5) with (8)). For example, to achieve 128 bits of statistical security, the optimal fixed C&C proportion requires 132 GCs, while the independent selection only requires 128 GCs; i.e., a reduction of about 3.0% in number of GCs.[12] The reduction is of 7.5% if considering a statistical security goal of 37 bits (40 vs. 37 GCs).

Fixed vs. variable C&C partition sizes. The S2PC-with-BitComs protocol defined in this paper accepts C&C methods for both fixed and variable partition sizes. The proof of security remains essentially the same – the error probabilities vary slightly but remain negligible in the number of GCs. Even though an independent selection of challenge types allows a lower number of GCs for the same error probability, some tradeoffs may make it preferable to have some control over the possible numbers of each type of challenge, namely considering that verification and evaluation challenges may have associated different communication and computation costs.

[12] In Table 1, method (8) actually requires that at least one index is selected for evaluation – this means that when both parties are honest the protocol always leads the parties to obtain the intended S2PC output (i.e., *perfect completeness*). In rigor, the error probability associated with an independent selection of challenges subject to this restriction corresponds to a value of statistical security that is negligibly lower than the assumed number of bits of statistical security, but to simplify the discussion this negligible difference is overlooked. The same observation applies to methods (6) and (7).

For example, if applying a *random seed checking* (RSC) type of technique [GMS08] (see details in §E), the communication cost associated with verification challenges is irrelevant, while the cost associated with *evaluation* challenges remains the same. Thus, if it is a priority to reduce the communication cost, then it may be preferable to reduce the number of evaluation challenges, at the expense of increasing the number of verification challenges (which may increase the computational cost of $\mathrm{P_B}$). Conversely, the computation necessary to evaluate a GC is typically lower than that required to verify the correctness of a GC. Thus, if communication is *cheap* and the computational capabilities of $\mathrm{P_B}$ are very reduced, then it might be preferable to ensure that there are more evaluation than verification challenges. It may also simply be considered a benefit to know in advance the exact amount of computation and communication that will take place, or at least be able to place some bounds. The above arguments show that it may be useful, in alternative to a (fully) independent selection of challenges (which is optimal in minimizing the number of GCs), to impose some restrictions on the number of verification and evaluation challenges.

Table 1 and Table 2 show some examples in more detail. For example, if it is simply preferable to ensure that the number of *evaluation* challenges is not more than half of the total challenges, then the challenge selection may remain independent except for that restriction (see (7)). For the same soundness error probability, this only requires one more GC in comparison with the full independent case (see Table 2). If it is useful to limit the number of *evaluation* challenges to a further lower number (e.g., to reduce communication cost), then a tradeoff can be made with the total number of GCs. For example, one may require that the number of evaluated GCs is at most one forth of the number of bits of statistical security, i.e., that each *evaluation* challenge provides at least 4 bits of security (6). As the "low e" row in Table 2 shows, the number of *evaluation* challenges can be ensured as intended, at the expense of increasing the number of GCs.

B Connectors

This section complements the explanation of how to implement the *connectors* that connect the decommitments of BitComs (of input and output bits) to the respective input and output wire keys of the GCs. It is worth emphasizing that connectors are only needed for the input and output wires of the circuit. Thus, the overhead brought by them does not depend on the number of intermediate circuit wires or gates in the circuit.

The connectors are used within the C&C approach, in a context somewhat similar to a commitment scheme that has a *commit* and a *reveal* phases. First, for each type of connector, the *setup* condition assumes that a BitCom has already been selected for the bit of the respective input or output wire, independently of the number of GCs. Then, in the *commit* phase, performed by $\mathrm{P_A}$ during the C&C COMMIT stage, $\mathrm{P_A}$ becomes bound to a connection between the BitCom decommitments and the respective wire keys. Then, in the RESPOND stage, $\mathrm{P_A}$ has to provide responses to either a *reveal for verification* or a *reveal for evaluation* mode. In the overall S2PC-with-BitComs protocol, $\mathrm{P_B}$ proceeds to the C&C EVALUATE stage only if all *connectors* have been it successfully verified, for the respectively selected *reveal* mode, and if the GCs selected for *verification* have also been successfully verified. It is worth noticing that connectors are related with the *initial* BitComs selected by the parties in the PRODUCE INITIAL BITCOMS stage, but not with the *final* BitComs obtained in the APPLY BITCOM PERMUTATIONS stage (after applying random permutations).

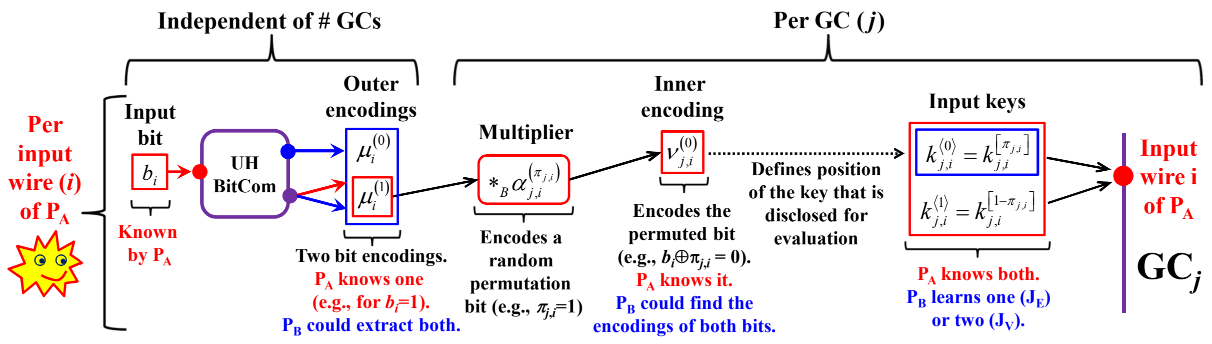

Independent of # GCs
Per GC (j)
Per input wire (i) of P_A
Input bit
b_i
Known by P_A
UH BitCom
Outer encodings
$\mu_i^{(0)}$
$\mu_i^{(1)}$
Two bit encodings. P_A knows one (e.g., for b_i=1). P_B could extract both.
Multiplier
$*_B \alpha_{j,i}^{(\pi_{j,i})}$
Encodes a random permutation bit (e.g., $\pi_{j,i}$=1)
Inner encoding
$\nu_{j,i}^{(0)}$
Encodes the permuted bit (e.g., $b_i \oplus \pi_{j,i} = 0$). P_A knows it. P_B could find the encodings of both bits.
Defines position of the key that is disclosed for evaluation
Input keys
$k_{j,i}^{\langle 0 \rangle} = k_{j,i}^{[\pi_{j,i}]}$
$k_{j,i}^{\langle 1 \rangle} = k_{j,i}^{[1-\pi_{j,i}]}$
P_A knows both. P_B learns one (J_E) or two (J_V).
Input wire i of P_A
GC_j

Fig. 4: **Scheme for input wires of P_A.** Legend: GC (*garbled circuit*); P_A (GC *constructor*); P_B (GC *evaluator*); i (input wire index); j (GC index); *class* of a group-element (bit that is encoded by the group-element); $\cdot^{(c)}$ (indication that a certain group element $\cdot$ is in class c); μ (*outer encoding* – encoding that decommits the private circuit input or output bit of a party, from a respective UH BitCom); α (*multiplier* – encoding of a permutation bit π, used only to permute a respective input bit of P_A); ν (*inner encoding* – encoding of permuted input bit of P_A); J_V and J_E (*verification* and *evaluation* subsets of challenge-indices); $*_p$ (XOR-homomorphic group-multiplication in the group that supports the BitComs whose trapdoor is known by P_p, for $p \in \{A, B\}$); $k^{[c]}$ (wire key with underlying bit c); $k^{\langle c \rangle}$ (wire key in position c within a pair of ordered wire keys).

B.1 Connectors for input of P_A

The *setup* condition is that P_A produces an UH BitCom of her own input bit, with trapdoor known by P_B. This is done with P_A selecting one encoding for her input bit and then sending the respective square to P_B. The input bit encoding is also denoted as *outer encoding*, in the sense that it stands outside of the connector, and in contrast with the *inner encoding* that is part of each connector and has some relation with the input keys of the GCs. Then, for each input wire of P_A in each GC, P_A decides two keys. For *evaluation* GCs, the main challenge is to ensure that, for each input wire, the bit underlying the key revealed by P_A is hidden from P_B and at the same time that the underlying bit is equal to the committed input bit. This is achieved with one connector for each input wire of P_A in each GC, connecting the (outer) encoding of the input bit of P_A with an (inner) encoding of the respective permuted bit. This connection is made via a multiplier that is itself an encoding of the respective permutation bit. The connector also includes the a respectively permuted pair of the two commitments of the two respective wire keys. In *verification* challenges, P_A decommits the permutation bit and reveals the two keys, allowing P_B to verify that the keys were consistently permuted. In *evaluation* challenges, P_B decommits the permuted bit and the key in the respective position. Some elements of the *connector* construction are depicted in Fig. 4, to ease the understanding of the description given in §3.2, and the more formal description in §C.

B.2 Connectors for input of P_B

For *evaluation* GCs, the challenge is that P_B must only learn one key per input wire, and P_A should not know which one was learned. Also, P_A should not be able to perform a selective failure attack (e.g., producing elements for which the evaluation by P_B is successful if a certain bit of P_B is 0, but unsuccessful if the bit is 1). The bit underlying each respective input key does not need to be hidden, both in the Verify and Evaluate stages, because P_B knows his own input bit. Thus, the random bit permutation considered for input wires of P_A does not need to be considered for the input wires of P_B.

The *setup* condition is that P_B selects one encoding for his input bit, and sends the respective square (i.e., the UH BitCom) to P_A. P_A knows the trapdoor of this 2-to-1

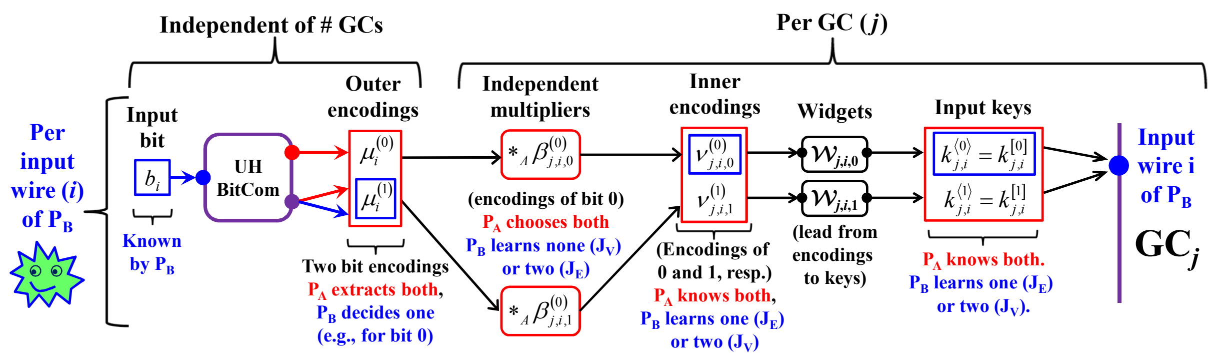


Fig. 5: **Scheme for input wires of P_B.** Legend: $\mathcal{W}$ (widget, converting group-elements into wire keys (for input wires of P_B), or converting wire keys into group elements (for output wires of P_B)); β (multiplier *class* 0, for input and output wires of P_B); see remaining items in the legend of Fig. 4.

square scheme and uses it to extract the two possible encodings from the BitCom value. Again, these bit encodings are dubbed *outer encodings*, to distinguish them from the inner encdings that will be made part of the connectors. Since the two decommitments of the BitCom are the trapdoor of P_A, they cannot be used directly as input keys, or otherwise P_B would learn the trapdoor once learning all the keys for verification GCs. The high level idea of the connector is to split the two non-trivially correlated (outer) encodings (the two possible decommitments of the input BitCom) into two independent (inner) encodings, so that P_B may learn one decommitment from each of them, instead of two from the original BitCom. Some elements of the *connector* construction are depicted in Fig. 5, to ease the understanding of the description given in §3.2, and the more formal description in §C.

Widgets – from inner encodings to input keys. In the initial description of the connectors for input wires of P_B (in §3.2), it was assumed that the inner encodings could be used as the keys of the respective input wire of the GC. If the garbling scheme allows deciding the input keys before generating the GC (i.e., give them as input to the garbling building mechanism), then the keys can be simply determined by applying a suitable transformation to the bit-encodings. For example, if input keys can be pseudo random bit-strings, then each key can be obtained as a suitable (pseudo-random) compressive commitment of the respective bit encoding. Nonetheless, it is possible to abstract more the garbling scheme. Specifically, it is possible to conceive garbling schemes where the input wire keys are part of the output of the GC generation and cannot be preset in advance. In such circumstance, P_A can still build a connector, by also sending to P_B the ciphertext resulting from encrypting the circuit input wire key, using the respective inner encoding as encryption key. In this scenario, whenever P_B learns one inner encoding for an input wire of P_A, it uses it as key to decrypt the ciphertext and obtain the respective circuit input key. The term *widget* is used to denote whatever construct is necessary to implement this conversion between inner encodings and wire keys, or vice-versa in the case of output wires.

B.3 Connectors for output of P_B

The connection between output wire keys and decommitments of output BitComs is similar to the case of input wires of P_B, except that it is made in the reverse order. For each output wire of P_B:

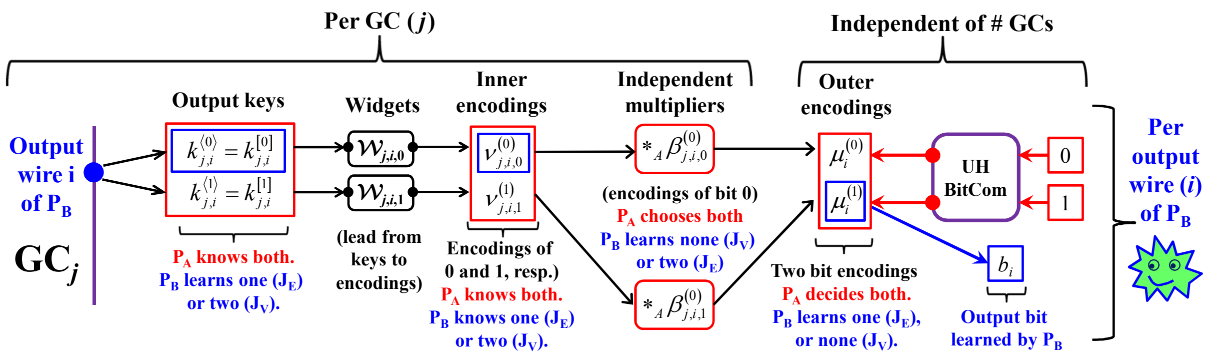

Per GC (*j*)
Independent of # GCs
Output wire i of P_B
GC_j
Output keys
$k^{\langle 0\rangle}_{j,i} = k^{[0]}_{j,i}$
$k^{\langle 1\rangle}_{j,i} = k^{[1]}_{j,i}$
P_A knows both. P_B learns one (J_E) or two (J_V).
Widgets
$\mathcal{W}_{j,i,0}$
$\mathcal{W}_{j,i,1}$
(lead from keys to encodings)
Inner encodings
$\nu^{(0)}_{j,i,0}$
$\nu^{(1)}_{j,i,1}$
Encodings of 0 and 1, resp.) P_A knows both. P_B knows one (J_E) or two (J_V).
Independent multipliers
$*_A\beta^{(0)}_{j,i,0}$
$*_A\beta^{(0)}_{j,i,1}$
(encodings of bit 0) P_A chooses both P_B learns none (J_V) or two (J_E)
Outer encodings
$\mu^{(0)}_i$
$\mu^{(1)}_i$
Two bit encodings P_A decides both. P_B learns one (J_E), or none (J_V).
UH BitCom
0
1
b_i
Output bit learned by P_B
Per output wire (*i*) of P_B

Fig. 6: **Scheme for output wires of P_B.** See legend items in the legends of Fig. 4 and Fig. 5.

- **Setup.** The initial BitCom is decided by P_A, even without any party yet knowing the respective circuit output bit. P_A simply selects a random pair of non-trivially correlated square-roots (decommitments, dubbed *outer encodings*) and sends the respective square (the UH BitCom) to P_B.
- **Commit.** Since the C&C VERIFY stage allows P_B to learn the two keys per output wire, the pair of outer encodings cannot directly be used as the circuit output keys – otherwise P_B would learn the trapdoor of P_A. P_A produces an ordered pair of independent bit encodings, dubbed *inner encodings*, encoding bits 0 and 1, respectively. P_A sends the pair of respective squares (dubbed inner squares) to P_B, serving as commitments, in a one-way sense, of the pair of inner encodings. For simplicity of description, it is assumed here that the inner encodings can be used directly as circuit output wire keys (this assumption can be relaxed using widgets).
- **Reveal for verification.** From the *verification* procedure of the GC, P_B simply obtains, for each output wire of P_B, the two output wire keys and respective underlying bits. P_B can verify that the two keys are proper square-roots of the previously received pair of inner squares and that they encode the proper bits; i.e., that (as group-elements, inner encodings) the bit they encode is equal to the bit that is underlying them (as wire-keys of the verified GC). P_B learns nothing about the bit encodings of the BitComs of the output bit, because the BitCom is independent.
- **Reveal for evaluation.** P_A reveals the two multipliers (encodings of bit 0) that lead the inner encodings into the respective outer encodings. P_B can verify that both multipliers are correct, by verifying that they both encode bit 0 and (due to the homomorphic properties) that their squares leads the two inner squares into the output BitCom. If something is found wrong here, then P_B can safely abort and complain, without jeopardizing its own privacy. If every verification is successful, P_B evaluates the GC (using the input keys obtained from the other types of connectors) in order to obtain one key per output wire. Finally, P_B verifies that the obtained output key is a proper square-root of one of the two inner squares and that it encodes the respective bit. This verification may fail, if P_A acted maliciously and was lucky that this GC was selected for evaluation and thus its connectors did not undergo the *reveal for verification* mode. In such case, P_B ignores all output associated with the GC, but does not complain (otherwise it would be vulnerable to a selective failure attack). Otherwise, if across all output wires the obtained output key is found valid (as inner encoding), then P_B combines it with the respective revealed multiplier to obtain the respective outer encoding, whih is a decommitment of the original output BitCom.

Widgets – from output keys to inner encodings. The assumption that the keys of output wires of GCs can be chosen group elements (inner encodings) was made to simplify the description. Nonetheless, the assumption can be relaxed with the use of *widgets*, symmetric in comparison with the widgets described for input wires of $\mathrm{P_B}$. Essentially, the connection between each output wire key and the respective inner encoding can be made by means of a ciphertext resulting from encrypting the inner encoding, using the respective output wire key as encryption key. A possible communication waste of this construction occurs whenever the size of group elements is significantly bigger than the size of wire keys (e.g., 3,072 bits for a Blum integer vs. 128 bits for a symmetric key). This can be improved by having the inner encodings (group elements) be obtained directly from the wire keys, using some pseudo-random generation procedure. In this case, it is assumed that each output key discloses the underlying bit, so that the pseudo-random-generation is able to generate a group element encoding the correct bit.

C Protocol specification

This section provides a more concrete specification of the new 1-output S2PC-with-BitComs protocol. The "1-output" characterization refers only to the goal of only one party ($\mathrm{P_B}$) learning a circuit output. In rigor, the protocol implements a probabilistic 2-output functionality, in the sense that the two parties receive random BitComs.

For a single page overview, the protocol is described in Fig. 8 using succinct notation (see notation in §H), making references to other sub-protocols, such as deciding the random C&C partition challenge (Fig. 7), deciding random BitCom permutations via a fully-simulatable two-party coin-tossing (§D, Fig. 9), and performing several ZKPoKs (§F, Figs. 13, 15, 17). A textual description is given below, to help interpret Fig. 8.

For simplicity, some aspects related with obvious syntactic or semantic verifications are left implicit; e.g., that received elements are within the expected domains, that mentioned square-roots in the context of an UH-BitCom scheme are meant to be proper square-roots, and that group operations undergo the respective necessary adjustments.

0. Setup

   In a setup phase, both parties agree on the goal of the protocol, namely what circuit (C) to securely evaluate and the roles of the two parties (who is the GC constructor ($\mathrm{P_A}$); and who is the GC evaluator ($\mathrm{P_B}$) that will learn the circuit output). The indices of positions of circuit input and output wires of both parties is also defined (43).

   The parties also agree on security parameters and instantiation of sub-protocols needed for the execution of the protocol (44):
   - A cryptographic security parameter κ and a statistical security parameter s', with which the protocol execution must conform.
   - A C&C partitioning method and respective parameters, consistent with the statistical security parameter. For example, the parties may want to define in advance the amount v of challenges that will be selected for *verification*, and the amount e of challenges that will be selected for *evaluation* (though not yet deciding which indices will be attributed which challenge type). In this case, the amounts must be selected in a way that the respective error probability of the protocol satisfies the number of intended bits of statistical security (see (4)). As another example, the parties may decide that there must be at least as many *verification* as *evaluation* challenges, and at least 1 *evaluation* challenge. In this case, the total number s of

challenges is defined as the smallest integer larger than the number s' of desired bits of statistical security (see (7)), and the values v and e remain undetermined until becoming defined in a coin-tossing in the later CHALLENGE stage of the protocol.

– The suite (ALGS) of necessary sub-protocols and algorithms:
  - The 2-to-1-square UH-BitCom scheme, the related UB-BitCom scheme, and respective parameters. The selection of parameters of BitCom schemes is left implicit – for example, for the instantiation based on Blum integers this would require each party to privately sample 2 primes, which can be done in advance and without interaction and re-used in different S2PC protocol executions with the same security parameter κ. Conversely, the ZKP of correctness of the parameters is considered explicitly – for example, for the instantiation based on Blum integers this can be achieved with a ZKPoK of Blum integer trapdoor of each 2-to-1 square scheme ((45), (46)).
  - Several ZKPoKs (see §F) related with BitComs.
  - The garbling scheme, containing functionalities to generate GCs (GC_{Build}), verify their correctness ($\text{GC}_{\text{Verify}}$) when in possession of enough information, and evaluate them (GC_{Eval}) when in possession of one input key per input wire.
  - A commitment scheme $\mathscr{C}$, defining the respective functionalities for commitment ($\mathscr{C}_{\text{Commit}}$) and verification ($\mathscr{C}_{\text{Verify}}$). The notation is adapted to a probabilistic scheme (which may be instantiated by an unconditionally binding, or unconditionally hiding, or computationally hiding and binding scheme). For instantiations using deterministic schemes, the random components (produced by $\mathscr{C}_{\text{Commit}}$ and used by $\mathscr{C}_{\text{Verify}}$) may simply be disregard.

1. PRODUCE INITIAL BITCOMS
   (a) **UH COMMIT INPUT BITS.** For each input wire, each party selects a random bit-encoding of her input bit (47), from the group for which the other party knows the trapdoor, and sends the respective square to the other party (48). This bit encoding is also denoted as *outer encoding*. P_B then gives a ZKPoK of decommitments of his UH BitComs, i.e., of square-roots of the squares related with his bits (49). P_A does not give the same ZKPoK, because it will do it ahead when considering also some UB BitComs. P_A uses her trapdoor to extract two bit encodings (outer encoding) from each UH BitCom of input bits of P_B ((50) and (51)). In this process, exponentiations are needed only for the extraction of square-roots. The complexity of other operations is dominated by modular multiplications. In particular, the ZKPoK of decommitments (ie., of square-roots) can be combined in parallel across all input wires of P_B (see Fig. 13), such that the number of communicated group-elements is linear in the statistical parameter.

   (b) **UB COMMIT INPUT BITS OF P_A.** P_A commits (encrypts) her own input bits with the UB-BitCom scheme whose trapdoor is the same as the one of the UH-BitCom scheme used by P_B to commit his own bits. Specifically, for each input bit of P_A, P_A selects a random group element (52) and then computes the encryption of the bit as the respective square or additive inverse of the square of the random group element, respective with the bit being a 0 or a 1 (53). Then, P_A gives a ZKPoK of equivalent decommitments of the UB and UH BitComs, i.e., proving that the known decommitments of the UB BitComs refer to the same bits as the (outer encodings) known decommitments of the UH BitComs (54) (see Fig. 15).

   (c) **UH COMMIT OUTPUT BITS OF P_B.** Within the 2-to-1 square scheme used for committing bits of P_B, P_A selects a random outer encoding of bit 0 for each output bit index of $\mathrm{P_B}$ (55) and then obtains the respective outer encoding for bit 1 by a simple multiplication with the non-trivial square-root of the group identity (i.e., the square-root in class 1 – for the instantiation based on Blum integers this is the square-root of 1 with Jacobi Symbol -1). Then, $\mathrm{P_A}$ sends the respective (outer) squares to $\mathrm{P_B}$ (these are the initial UH BitComs of the output bits of $\mathrm{P_B}$, even though the respective bits have not yet been computed) (56).

2. Commit
   $\mathrm{P_A}$ sends the GCs and respective connectors to $\mathrm{P_B}$, as follows. For each of the s GC indices:
   - **GCs.** $\mathrm{P_A}$ uses the garbling $\mathrm{GC_{Build}}$ probabilistic functionality to produce a GC and respective input and output wire keys (57), along with any extra randomness that might be needed to later verify the correctness of the GC. As input, $\mathrm{P_A}$ uses the Boolean circuit specification (C) and the cryptographic security parameter (κ). $\mathrm{P_A}$ keeps the input and output wire keys secret, as well as the extra randomness, but sends the GC to $\mathrm{P_B}$ (58).
   - **Connectors input $\mathbf{P_A}$.** $\mathrm{P_A}$ commits to the connector of each of her input wires as follows. It selects a random bit permutation and a respective encoding (dubbed *multiplier*) (59) within the 2-to-1-square scheme with trapdoor known by $\mathrm{P_B}$. Then it computes the homomorphic composition of the outer encoding (the decommitment of the input BitCom) with the multiplier, thus obtaining an inner encoding that encodes the permuted version of her input bit (60). $\mathrm{P_A}$ sends the square of the inner encoding to $\mathrm{P_B}$, as an UH BitCom of the permuted version of her input bit (61). $\mathrm{P_A}$ then produces a commitment of each of the two circuit input wire keys, along with the randomness needed to verify them (62).[13] Then, $\mathrm{P_A}$ joins the two commitments into a pair permuted by the previously decided random bit-permutation ((63) and (64)), and sends the permuted pair to $\mathrm{P_B}$.
   - **Connectors input+output $\mathbf{P_B}$.**
     - **For each input wire of $\mathbf{P_B}$**, and for each possible bit value, $\mathrm{P_A}$ selects a random bit-encoding of 0 (dubbed *multiplier*) and composes it with the outer encoding (the decommitment that encodes the respective bit value), to obtain a new *inner encoding*. In this way, $\mathrm{P_A}$ obtains two independent inner encodings, one for bit 0 and the other for bit 1, for each input wire of $\mathrm{P_B}$ (65). If widgets are necessary, to connect these inner encodings to the respective input wire keys, $\mathrm{P_A}$ sends them to $\mathrm{P_B}$ (66).
     - **For each output wire of $\mathbf{P_B}$**, and for each possible bit value, $\mathrm{P_A}$ selects a random bit-encoding of 0 (dubbed *inverse multiplier*) and composes it with the outer encoding (the BitCom decommitment that encodes the respective bit value), to obtain a new *inner encoding* (67). The notation was changed from *multiplier* to *inverse multiplier*, so that the direction of the *multipliers* is indeed symmetric between the input and output wires of $\mathrm{P_B}$ (compare Fig. 5 with Fig. 6). More specifically, later in the Evaluate stage, the multiplicative inverse of the *inverse multiplier*, which will be simply dubbed as *multiplier*, will be used to lead the inner encodings into the outer encodings. If widgets are necessary, to connect the output wire-keys to the respective new bit-encodings, $\mathrm{P_A}$ sends them to $\mathrm{P_B}$ (68).

[13] For generality, the commitment is assumed here be hiding and binding. In practice, if the garbling scheme security is maintained as long as $\mathrm{P_B}$ cannot recover input keys, even if it recovers a few bits of the keys, then the commitment can be simply based on a suitable one-way and collision resistant hash function.

- **For each input and output wire of $\mathrm{P_B}$**, $\mathrm{P_A}$ sends the square of each of the two inner encodings to $\mathrm{P_B}$ (69), serving as commitments (in a one-way sense) of the respective inner encodings.

– **Note on widgets**: The description based on widgets is presented for the sake of generality, but there are efficient garbling schemes that allow avoiding them. This is the case if the mechanism to build GCs ($\mathrm{GC_{Build}}$) accepts a pre-definition of circuit input keys and circuit output keys. In such case, the widgets can be a simple specification of pseudo-random generation procedures (thus defined directly in the Setup stage), adapted to output either wire keys or group elements of a certain class, respectively for input and output wires of $\mathrm{P_B}$.

3. Challenge

$\mathrm{P_A}$ and $\mathrm{P_B}$ engage in a sub-protocol to decide a random vector of challenge bits (70), conditioned to the method agreed in the Setup stage. Once the vector is decided, the positions with bit 1 represent *verification* challenges, and the remaining positions represent *evaluation* challenges. Within the overall protocol, this coin-tossing needs to allow a simulated $\mathrm{P_A}$, with rewindable access to a possibly malicious $\mathrm{P_B^*}$, to decide the outcome of the coin-tossing whenever $\mathrm{P_B^*}$ does not abort.

As an example, the selection of challenges may be conditioned in the number of verification challenges. The exemplified sub-protocol do decide the random C&C partition, described using succinct notation in Fig. 7, considers two possible modes: a Fixed mode, where the number of verification challenges is pre-determined; and a Variable mode, where the number of verification challenges is conditioned to be in a certain interval. Other than these restrictions, the C&C vector is obtained uniformly from the set of admissible vectors, which means that the number of verification challenges is not uniformly distributed over the possible values (the most likely values are those closer to half the number of challenges). A textual description of the sub-protocol follows:

– **Goal and common input.** The high level goal of this sub-protocol is to decide a predetermined number of random challenge bits (9), which will determine the C&C partition. These bits will form a *vector* of bit challenges, with the positions of bit 0 denoting *verification* challenges, and the positions of bit 1 denoting *evaluation* challenges. The input of the protocol contains a pair of integers (10) defining the interval of possible numbers of verification challenges. If the interval contains only one possible value (11), then the C&C mode is defined as Fixed, and naturally the number of verification challenges becomes fixed (12). If the interval contains different numbers, then the mode is defined as Variable (13). If in this Variable mode, the parties count the number of possible partitions (14) and then define the number of temporary bits that need to be coin-tossed so that it is possible (with overwhelming probability in the statistical security parameter) to select a random partition uniformly (15). For clarity, in the following explanation these bits will be referred as part of a *bit-string*, whereas the bit challenges will be referred as being part of a *vector* of challenges. The protocol also requires usage of a commitment scheme that is at least computationally hiding and biding (16).

– **Commit contribution of $\mathbf{P_B}$.**

- **If in the Fixed mode.** $\mathrm{P_B}$ starts by selecting a random vector as intended for the outcome of the protocol, i.e., with as much bits as the number of challenges, and conditioned to the fixed number of *verification* challenges (i.e., with exactly a certain number of 0's) (17). $\mathrm{P_B}$ defines this vector as his *contribution* that needs to be committed (18).

- **If in the** Variable **mode.** $\mathrm{P_B}$ selects a random bit-string (19) with the length previously determined (see (15)) and also a random permutation of the positions of the final vector of bit challenges that will be determined (20). $\mathrm{P_B}$ pairs these two values into his *contribution* that needs to be committed (21).

$\mathrm{P_B}$ computes a commitment to his contribution (22) and sends the respective public part to $\mathrm{P_A}$ 23.

– **Contribution of $\mathbf{P_A}$.** $\mathrm{P_A}$ selects a random permutation of the positions of the challenge vector (still to be determined) and sends it to $\mathrm{P_B}$ (24). Then, if in the Variable mode, $\mathrm{P_A}$ also selects a random bit-string with the predetermined number of bits and sends it to $\mathrm{P_B}$ (25).

```
Setup – common input. $\mathrm{P_A}, \mathrm{P_B}$ :
  $s$ (# of GCs, i.e., # of bit-challenges to decide)   (9)
  $v' \in \left\{(v_0, v_1) \in \{0,...,s\}^2 : 0 \leq v_0 < v_1 \leq s\right\}$   (10)
    (defines method for selection of challenges)
    If $v_0 =^? v_1$, then mode=Fixed   (11)
      $v = v_0$ (fixed partition sizes)   (12)
    else mode=Variable (variable partition sizes)   (13)
      $t \equiv \sum_{i=v_0,...,v_1} \mathrm{Bin}(s,i); |t| \equiv \lceil \log_2(t-1) \rceil$   (14)
      $m \equiv |t| + s'$ (size of bit-string to generate)   (15)
  $\mathscr{C}$ (at least computationally hiding and binding)   (16)
Commit contribution of $\mathrm{P_B}$. $\mathrm{P_B}$ :
  If mode $=^?$ Fixed :
    $\sigma^{(B)} \leftarrow^{\$} \{\sigma' \in \{0,1\}^s : \#(\{j : \sigma'_j = 0\}) = v\}$   (17)
    $\theta = \sigma^{(B)}$   (18)
  If mode $=^?$ Variable :
    $\sigma^{(B)} \leftarrow^{\$} \{0,1\}^m$ (size of bit-string to generate)   (19)
    $\rho^{(B)} \leftarrow^{\$} \mathrm{Perms}(\langle 1,...,s \rangle)$ (permutation)   (20)
    $\theta = \left\langle \sigma^{(B)}, \rho^{(B)} \right\rangle$   (21)
  $\mathrm{P_B} : \langle \underline{\theta}, \bar{\theta} \rangle \leftarrow^{\$} \mathscr{C}[\theta]$   (22)
  $\mathrm{P_B} \rightarrow \mathrm{P_A} : \bar{\theta}$ (commitment)   (23)
Contribution of $\mathrm{P_A}$.
  $\mathrm{P_A} \rightarrow \mathrm{P_B} : \rho^{(A)} \leftarrow^{\$} \mathrm{Perm}(\langle 1,...,s \rangle)$   (24)
  If mode $=^?$ Variable :
    then $\mathrm{P_A} \rightarrow \mathrm{P_B} : \sigma^{(A)} \leftarrow^{\$} \{0,1\}^m$   (25)
Reveal contribution of $\mathrm{P_B}$.
  $\mathrm{P_B} \rightarrow \mathrm{P_A} : \langle \theta, \underline{\theta} \rangle$ (decommitment)   (26)
  Parse decommitted value.
    $\mathrm{P_A}$ : If $\neg \mathscr{C}_{\mathrm{Verify}}(\mathscr{C}, \theta, \underline{\theta}, \bar{\theta})$, then Abort   (27)
    $\mathrm{P_A}$ : If mode $=^?$ Fixed : $\sigma^{(B)} = \theta$   (28)
    $\mathrm{P_A}$ : If mode $=^?$ Variable : $\left\langle \sigma^{(B)}, \rho^{(B)} \right\rangle = \theta$   (29)
Verifications.
  $\mathrm{P_B}$ : If $\rho^{(A)} \notin \mathrm{Perm}(\langle 1,...,s \rangle)$, then Abort   (30)
  If mode $=^?$ Fixed :   (31)
    $\mathrm{P_A}$ : If $\sigma^{(B)} \notin \{0,1\}^s \vee \#(j : \sigma^{(B)}_j = 0) \neq v$, then Abort
  If mode $=^?$ Variable :
    $\mathrm{P_A}$ : If $\sigma^{(B)} \notin \{0,1\}^m$ , then Abort   (32)
    $\mathrm{P_A}$ : If $\rho^{(B)} \notin \mathrm{Perm}(\langle 1,...,s \rangle)$, then Abort   (33)
    $\mathrm{P_B}$ : If $\sigma^{(A)} \notin \{0,1\}^m$ , then Abort   (34)
Decide final C&C challenge partition. $\mathrm{P_A}, \mathrm{P_B}$ :
  If mode $=^?$ Fixed :
    $\chi = \rho^{(A)}(\sigma^{(A)})$   (35)
    $J_V = \{j : \chi_j = 0\}, J_E = \{j : \chi_j = 1\}$   (36)
  If mode $=^?$ Variable :
    $\sigma = \sigma^{(A)} \oplus \sigma^{(B)}, \rho = \rho^{(A)} \circ \rho(B)$   (37)
    $r = \mathrm{GetInt}_{\leq t}(\sigma)$ (random int $\leq t$)   (38)
    $v = min\left\{u = v_0,...,v_1 : r \leq \sum_{i=v_0}^{u} \mathrm{Bin}(s,i)\right\}$   (39)
    $J_V = \{\rho_i : i = 1,...,v\}$   (40)
    $J_E = \{\rho_i : i = v+1,...,s\}$   (41)
  $\mathrm{P_A}, \mathrm{P_B} \downarrow \langle J_V, J_E \rangle$ (output)   (42)
```

Fig. 7: **Protocol – decide random cut-and-choose partition.** This protocol is not fully-simulatable – for the purpose of the S2PC protocol, it is only required that the simulator (impersonating $\mathrm{P_A}$) with rewindable black-box access to $\mathrm{P_B^*}$ is capable of leading the coin-tossing to the intended value, whenever $\mathrm{P_B^*}$ does not abort. The simulator proceeds until the step where $\mathrm{P_B^*}$ reveals his contribution, and then verifies that the decommitment is correct (26). Then, the simulator rewinds and changes the contribution of $\mathrm{P_A}$ to the value necessary to lead the final outcome to be the intended value. Within the overall S2PC protocol, an *abort* action by a malicious $\mathrm{P_B^*}$ in this coin-tossing sub-protocol is not a problem, because the simulator only needs to lead the outcome to be a certain value after $\mathrm{P_B^*}$ has already shown some willingness in allowing such value (i.e., if at least once the protocol has previously resulted in such value). Thus, at most the simulator might need to rewind further back, allowing $\mathrm{P_B}$ to commit again to some (possibly other) contribution, and restart the procedure.

- **Reveal contribution of $\mathrm{P_B}$.** $\mathrm{P_B}$ decommits his contribution (26), and $\mathrm{P_A}$ verifies that the decommitment is consistent with the respective commitment (27). $\mathrm{P_A}$ parses the contribution according to the mode of operation. Specifically, if in the FIXED mode, then it expects to obtain a vector of challenges (28). If in the VARIABLE mode, then it expects to obtains a bit-string of the predetermined length and also permutation of the vector of challenges (29).
- **Verifications.** $\mathrm{P_A}$ and $\mathrm{P_B}$ make the necessary verifications to the contributions received from the other party (see (30), (31), (32), (33), (34)).
- **Decide final C&C challenge partition.** Finally, each party computes locally the C&C partition, based on the contributions from both parties.
  - **If in the** FIXED **mode.** The vector of challenges is obtained by applying the permutation contributed by $\mathrm{P_A}$ to the vector contributed by $\mathrm{P_B}$ (35). As mentioned, the subsets of verification and evaluation challenges consist on the positions for which the final (permuted) vector has 0's and 1's, respectively (36).
  - **If in the** VARIABLE **mode.** The parties compute the XOR of the bit-strings contributed by both parties, to obtain a random string of the same length. The parties combine the permutations contributed by both parties to obtain a random permutation of the length of the vector of challenges (37). From the bit-string, the parties retrieve a random positive integer lower then the number of possible C&C configurations (38). From this integer, the parties decode the final number of verification challenges (39). Finally, the subset of indices of verification challenges is defined as the values in the left positions of the random permutation (40), up to the amount of verification challenges, and the subset of indices of evaluation challenges is defined as the remaining values in the right positions of the random permutation (41).[14] Both parties decide the two subsets as the output of the sub-protocol (42).

4. DECIDE UH-BITCOM PERMUTATIONS
   $\mathrm{P_A}$ and $\mathrm{P_B}$ engage in a sub-protocol to decide random permutations (encodings of 0 in the respective groups) for all input and output wires (71), with $\mathrm{P_A}$ being the first one to learn the outcome of the coin-tossing (this priority by $\mathrm{P_A}$ can change, according to the ideal functionality defined in §G.1). §D explains the requirements of this sub-protocol, namely about full-simulability.
5. RESPOND
   - **Verification indices.** $\mathrm{P_A}$ proceeds with the *reveal for verification* phase of the connectors associated with GCs selected for verification. For each challenge index:
     - $\mathrm{P_A}$ reveals the extra randomness needed to verify the correctness of the GC (72).
     - For input wires of $\mathrm{P_A}$, $\mathrm{P_A}$ reveals the multiplier (the bit encoding of the permutation bit) and decommits the two circuit input keys in non-permuted order, by revealing the keys and extra randomness that might be needed to verify them (73).
     - For input wires of $\mathrm{P_B}$, $\mathrm{P_A}$ decommits the two independent inner encodings (the first being of bit 0 and the second being of bit 1) that are square-roots of the two inner squares (74).

[14] Actually, a random C&C vector could be directly obtained from the random bit-string, but the description based on an additional permutation of vector positions might be more intuitive for the reader.

- **Evaluation indices.** P_A proceeds with the *reveal for evaluation* phase of the connectors associated with GCs selected for verification.
  - For input wires of P_A: P_A decommits the permuted bit, by revealing the inner encoding (i.e., the encoding of the permuted bit) (75); P_A also decommits the circuit input key in the respective permuted position (i.e., with the underlying bit being the input bit of P_A), by revealing the key and extra randomness that might be needed to verify it (76). Since P_B does not know the respective underlying bit, at this point the key is called a tentative (evaluation) key, as a way to avoiding referring which of the two keys it should be.
  - For input wires of P_B, P_A reveals the two multipliers, both encoding bit 0 (77). These multipliers connect the two outer encodings (non-trivially correlated encodings of the initial BitComs, of which P_B only knows one) to the two inner encodings. It is worth noticing that the direction of the connection is opposite in the case of output wires. Specifically, for output wires the respective multipliers lead from inner to outer encodings (of output bits).

6. Verify
   Using the obtained responses, P_B verifies the GCs and connectors, as follows.
   - **For each verification index:**
     - **Connectors input P_A.** P_B checks that the square of the revealed multiplier leads the input BitCom into the inner square (78) that was received earlier as BitCom of the permuted bit (61), and determines the encoded bit of the multiplier as being the permutation bit (79). For each input key assumed to have a certain underlying bit, P_B verifies it against the key-commitment in the respective permuted position (80).
     - **Connectors input P_B.** P_B checks that the two received inner encodings indeed encode bits 0 and 1, respectively, and that their squares are consistent with the inner squares (81) that were received earlier (69). P_B then uses the widgets to obtain the circuit input keys from the respective inner encodings (82).
     - **GCs.** In possession of all input wires keys of the GC selected for verification (83), and the extra randomness necessary to verify the GC, P_B uses the garbling verification procedure (GC_{Ver}) to verify that the GC is indeed correct (84). In particular, the verification is successful if and only if the GC is consistent with the intended Boolean circuit and the circuit input keys are correct.
     - **Connectors output P_B.** Since P_B has all input keys, it uses them to obtain all output keys and respective underlying bits from the GC (85). Then, from each output key (with known underlying bit), P_B uses the respective widget to obtain the respective inner encoding (86). Finally, P_B verifies that the obtained inner encodings are encodings of the expected bits and are valid square-roots of the inner squares (87) that were received earlier (69).
   - **For each evaluation index:**[15]
     Even for *evaluation* indices, P_B still makes verifications related with the *reveal for evaluation* phase of the connectors.
     - **For input wires of P_A:** P_B computes the bit encoded by revealed inner encoding (which P_A claims to encode the permuted input bit of P_A), and verifies that the inner encoding is indeed a proper square-root of the inner square (88) that was received earlier (61). P_A also verifies the decommitted (tentative) input key against the respective key-commitment in the permuted position (89).

[15] Even though these verifications are related with *evaluation* indices (i.e., indices that contain the GCs that will be evaluated), they do not depend on the input bits of P_B and thus do not allow a selective failure attack.

- **For input wires of $\mathrm{P_B}$:** $\mathrm{P_B}$ verifies that the two received multipliers encode 0, and that their squares lead the initial BitCom of the input bit into the two inner squares (90) that were received earlier (69).
- **For output wires of $\mathrm{P_B}$:** $\mathrm{P_B}$ verifies that the two received multipliers encode 0, and that their squares lead the 2 inner squares (that were received earlier (69)) into the initial BitCom of the output bit (91).

If any verification above has failed, then $\mathrm{P_B}$ aborts, outputting Fail.

7. Evaluate

$\mathrm{P_B}$ initializes an an empty list, dubbed Ignore list (92).

- **Evaluate GCs.** For each *evaluation* index: for each input wire index of $\mathrm{P_B}$, $\mathrm{P_B}$ multiplies the known outer encoding (the BitCom decommitment that encodes the private input bit of $\mathrm{P_B}$) with the respective received multiplier (93), thus obtaining a respective inner encoding; then, $\mathrm{P_B}$ applies the respective widget to obtain the respective (tentative) input wire key (94); in this way, $\mathrm{P_A}$ learns one tentative key per circuit input wire of $\mathrm{P_A}$ and $\mathrm{P_B}$ (95); $\mathrm{P_A}$ uses the keys obtained across all input wires to evaluate the GC and thus obtain one tentative key per circuit output wire (96).
- **Get output bit encodings.** For simplicity of description, it is assumed here that each output key reveals the respective underlying bit, e.g., its least significant bit ($\epsilon(\cdot)$).[16] For each output wire index, in each GC: $\mathrm{P_B}$ applies the respective output widget to the obtained output key, in order to obtain a tentative inner encoding (97). $\mathrm{P_B}$ then verifies that the inner encoding encodes the expected bit and that its square coincides with the respective inner square (98) that was received earlier (69). If this is the case, $\mathrm{P_B}$ computes the outer encoding (decommitment of the respective output BitCom) (99), by multiplying the inner encoding with the respective multiplier that was received earlier (77) and verified for correctness (90). Otherwise, if the tentative inner encoding is not valid, then $\mathrm{P_B}$ adds the index to the Ignore list, and ignores this index henceforth (100).[17]
- **Check for inconsistencies.** For each output wire index, $\mathrm{P_B}$ gathers all valid pairs of bit and respective encoding received across the several evaluation GCs (101). If for every output wire index there are no inconsistencies across different GCs (i.e., if for each wire index all pairs are the same) (102), then $\mathrm{P_B}$ simply accepts these bits and respective bit encodings as correct (103). If $\mathrm{P_A}$ finds instead that for some output wire there is more than one outer encoding (i.e., decommitment) found for the respective BitCom, then $\mathrm{P_B}$ activates the forge-and-lose path (104) to recover the final outputs. Within this path, $\mathrm{P_B}$ extracts the trapdoor of $\mathrm{P_A}$ as a non-trivially-correlated pair of proper square-roots (105), uses it to obtain the input of $\mathrm{P_A}$ from the respective UB BitComs (106), then computes in the clear the circuit output bits (107) and finally chooses the decommitments that encode the correct output bits (108).[18]

[16] The simplification is in allowing $\mathrm{P_B}$ to directly know which widget to use, but the assumption is not essential. If the garbling scheme does not allow it, then the information can be, for example, embedded in the widgets.

[17] Actually, the index is ignored in terms of its contribution for the values determined in the remainder of the procedure, but in practice the index may have to be accounted in counter-measures put in place to avoid side channel attacks. For example, a malicious $\mathrm{P_A}$ must not be able to find whether or not there are indices which were *ignored*.

[18] The encodings could instead be extracted with the trapdoor, but that would require more exponentiations.

0. SETUP. $\mathrm{P_A} \leftrightarrow \mathrm{P_B} : (C \equiv C_B, (I_\mathrm{A}, I_B, \mathrm{O_A} \equiv \oslash, \mathrm{O_B}))$ (43)

$$\mathrm{P_A} \leftrightarrow \mathrm{P_B} : \left(\left(1^\kappa, 1^{s'}\right), (s, v, e), \mathrm{ALGS}\right) \tag{44}$$

$$\mathrm{P_A}(t_A) \leftrightarrow \mathrm{P_B} : \mathrm{ZKPoK}_{\text{BI-trap}}(\mathbb{G}_A) \text{ (see Fig. 17)} \tag{45}$$

$$\mathrm{P_B}(t_B) \leftrightarrow \mathrm{P_A} : \mathrm{ZKPoK}_{\text{BI-trap}}(\mathbb{G}_B) \text{ (see Fig. 17)} \tag{46}$$

1. PRODUCE INITIAL BITCOMS.

UH COMMIT INPUT BITS.

$$\mathrm{P}_p : \mu_i \equiv \mu_i^{(b_i)} \leftarrow^\$ h_{\bar{p}}^{-1}(b_i) : i \in I_p : p \in \{A, B\} \tag{47}$$

$$\mathrm{P}_p \rightarrow \mathrm{P}_{\bar{p}} : \mu_i' = (\mu_i^{(b_i)})_{\bar{p}}^2 : i \in I_p : p \in \{A, B\} \tag{48}$$

$$\mathrm{P_B}(\mu_{I_B}) \leftrightarrow \mathrm{P_A} : \mathrm{ZKPoK}_{\mathrm{Sqrts}}(\mu'_{I_B}) \text{ (see Fig. 13)} \tag{49}$$

$$\mathrm{P_A} : \mu_i^{(0)} = \mathrm{SQRT}\,[t_A]_A^{(0)}\,(\mu_i') : i \in I_B \tag{50}$$

$$\mathrm{P_A} : \mu_i^{(1)} = \mu_i^{(0)} *_A \mathrm{NTSQRT1}_A : i \in I_B \tag{51}$$

UB COMMIT INPUT BITS OF $\mathrm{P_A}$.

$$\mathrm{P_A} : \mathcal{U}_i \leftarrow^\$ \mathbb{G}_A : i \in I_\mathrm{A} \tag{52}$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : \mathcal{U}_i' = (-1)^{b_i} *_A (\mathcal{U}_i)_A^2 : i \in I_\mathrm{A} \tag{53}$$

$\mathrm{P_A}(\mu_{I_\mathrm{A}}, \mathcal{U}_{I_\mathrm{A}}) \leftrightarrow \mathrm{P_B}$:

$$\mathrm{ZKPoK}_{\mathrm{Equiv}}(\mu'_{I_\mathrm{A}}, \mathcal{U}'_{I_\mathrm{A}}) \text{ (see Fig. 15)} \tag{54}$$

UH COMMIT OUTPUT BITS OF $\mathrm{P_B}$. For $i \in \mathrm{O_B}$:

$$\mathrm{P_A} : \mu_i^{(0)} \leftarrow^\$ h_A^{-1}(0);\ \mu_i^{(1)} = \mu_i^{(0)} *_A \mathrm{NTSQRT1}_A \tag{55}$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : \mu' = \left(\mu_i^{(0)}\right)_A^2 \tag{56}$$

2. COMMIT. For $j \in \{1, ..., s\}$:

GCs. $\mathrm{P_A} : \langle GC_j, \mathrm{IOkeys}_j, r_j\rangle \leftarrow^\$ \mathrm{GC_{Build}}[C, \kappa]$ (57)

$$(\mathrm{IOkeys}_j \equiv \left\langle (i, c, k_{j,i}^{[c]}) : i \in I_{A,B} \cup \mathrm{O_B}, c \in \{0, 1\}\right\rangle)$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : GC_j \tag{58}$$

Connectors input $\mathbf{P_A}$. For $i \in I_\mathrm{A}$:

$$\mathrm{P_A} : \pi_{j,i} \leftarrow^\$ \{0, 1\};\ \alpha_{j,i} \leftarrow^\$ h_B^{-1}(\pi_{j,i}) \tag{59}$$

$$\mathrm{P_A} : \nu_{j,i}^{[b_i]} = \mu_i^{(b_i)} *_B \alpha_{j,i} \tag{60}$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : {\nu'}_{j,i} = (\nu_{j,i})_B^2 \tag{61}$$

$$\mathrm{P_A} : \left(\underline{k}_{j,i}^{[c]}, \bar{k}_{j,i}^{[c]}\right) \leftarrow^\$ \mathscr{C}_{\mathrm{Commit}}\left[k_{j,i}^{[c]}\right] : c \in \{0, 1\} \tag{62}$$

$$\mathrm{P_A} : \langle c\rangle \equiv [c \oplus \pi_{j,i}] \tag{63}$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : \left(\bar{k}_{j,i}^{\langle 0\rangle}, \bar{k}_{j,i}^{\langle 1\rangle}\right) \tag{64}$$

Connectors input $\mathbf{P_B}$. For $i \in I_B$ and $c \in \{0, 1\}$:

$$\mathrm{P_A} : \beta_{j,i,c} \leftarrow^\$ h_A^{-1}(0), \quad \nu_{j,i,c}^{[c]} = \mu_i^{[c]} *_A \beta_{j,i,c} \tag{65}$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : \mathcal{W}_{j,i,c} = \textsc{Widget}\left[\nu_{j,i,c} \rightarrow k_{j,i}^{[c]}\right] \tag{66}$$

Connectors output $\mathbf{P_B}$. For $i \in \mathrm{O_B}$ and $c \in \{0, 1\}$:

$$\mathrm{P_A} : (\beta_{j,i,c})^{-1} \leftarrow^\$ h_A^{-1}(0), \quad \nu_{j,i,c}^{[c]} = \mu_i^{[c]} *_A (\beta_{j,i,c})^{-1} \tag{67}$$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : \mathcal{W}_{j,i,c} = \textsc{Widget}\left[k_{j,i}^{[c]} \rightarrow \nu_{j,i,c}\right] \tag{68}$$

Squares of $\mathbf{P_B}$. For $i \in IO_\mathrm{B}$ and $c \in \{0, 1\}$

$$\mathrm{P_A} \rightarrow \mathrm{P_B} : {\nu'}_{j,i,c} = (\nu_{j,i,c})_A^2 \tag{69}$$

3. CHALLENGE.

$$\mathrm{P_B} \leftrightarrow \mathrm{P_A} : \text{decide } (J_V, J_E) \text{ (see Fig. 7)} \tag{70}$$

4. DECIDE UH-BITCOM PERMUTATIONS. For $p \in \{A, B\}$:

$$\mathrm{P_A} \leftrightarrow \mathrm{P_B} : \gamma_i \leftarrow^\$ h_{\bar{p}}^{-1}(0) : i \in IO_p \text{ (see §D)} \tag{71}$$

5. RESPOND. $\mathrm{P_A} \rightarrow \mathrm{P_B}$:

***Verification* indices.** For $j \in J_V$:

$$r_j \text{ (extra randomness needed to verify GC)} \tag{72}$$

$$\alpha_{j,i}, \left(\left(k_{j,i}^{[0]}, \underline{k}_{j,i}^{[0]}\right), \left(k_{j,i}^{[1]}, \underline{k}_{j,i}^{[1]}\right)\right) : i \in I_\mathrm{A} \tag{73}$$

$$\left(\nu_{j,i,0}^{[0]}, \nu_{j,i,1}^{[1]}\right) : i \in I_B \tag{74}$$

***Evaluation* indices.** For $j \in J_E$:

$$\nu_{j,i}^{(b_i \oplus \pi_{j,i})} \equiv u_i^{[b_i]} *_B \alpha_{j,i} : i \in I_\mathrm{A} \tag{75}$$

$$\left(\xi_{j,i}, \underline{\xi}_{j,i}\right) \equiv \left(k_{j,i}^{\langle b_i \oplus \pi_{j,i}\rangle}, \underline{k}_{j,i}^{\langle b_i \oplus \pi_{j,i}\rangle}\right) : i \in I_\mathrm{A} \tag{76}$$

$$(\beta_{j,i,0}, \beta_{j,i,1}) : i \in IO_\mathrm{B} \tag{77}$$

6. VERIFY. $\mathrm{P_B}$:

***Verification* indices.** For $j \in J_V$:

Input of $\mathbf{P_A}$. For $i \in I_\mathrm{A}$:

$${\mu'}_{j,i} *_B (\alpha_{j,i})_B^2 =^? \nu'_{j,i} \tag{78}$$

$$\pi_{j,i} = h_B(\alpha_{j,i}) \tag{79}$$

$$\mathscr{C}_{\mathrm{Verify}}\left(k_{j,i}^{[c \oplus \pi_{j,i}]}, \underline{k}_{j,i}^{[c \oplus \pi_{j,i}]}, \bar{k}_{j,i}^{\langle c\rangle}\right) : c \in \{0, 1\} \tag{80}$$

Input of $\mathbf{P_B}$. For $i \in I_B$ and $c \in \{0, 1\}$:

$$h_A\left(\nu_{j,i,c}^{[c]}\right) =^? c \wedge (\nu_{j,i,c})_A^2 =^? \nu'_{j,i,c} \tag{81}$$

$$k_{j,i}^{[c]} = \mathcal{W}_{j,i}\left(\nu_{j,i,c}^{[c]}\right) \tag{82}$$

GCs. For $j \in J_V$:

$$\mathrm{InKeys}_j^{(2)} \equiv \left\langle \left(i, c, k_{j,i}^{[c]}\right) : i \in I_{A,B}, c \in \{0, 1\}\right\rangle \tag{83}$$

$$GC_{\mathrm{Verify}}\left(GC_j, \mathrm{InKeys}_j^{(2)}, r_j, C, \kappa\right) \tag{84}$$

Output of $\mathbf{P_B}$. For $i \in \mathrm{O_B}$ and $c \in \{0, 1\}$:

$$\mathrm{OutKeys}_j^{(2)} = \mathrm{GC}_{\text{Get-OutKeys}}\left(GC_j, \mathrm{InKeys}_j^{(2)}\right) \tag{85}$$

$$(\mathrm{OutKeys}_j^{(2)} \equiv \left\langle (i, c, k_{j,i}^{[c]}) : i \in \mathrm{O_B}, c \in \{0, 1\}\right\rangle)$$

$$\mathrm{P_B} : \nu_{j,i,c} = \mathcal{W}_{j,i}(k_{j,i}^{[c]}) \tag{86}$$

$$h_A(\nu_{j,i,c}) =^? c \wedge (\nu_{j,i,c})_A^2 =^? \nu'_{j,i,c} \tag{87}$$

***Evaluation* indices.** For $j \in J_E$:

$$c_{j,i} = h_B(\nu_{j,i}); \quad (\nu_{j,i})_B^2 =^? \nu'_{j,i} : i \in I_\mathrm{A} \tag{88}$$

$$\mathscr{C}_{\mathrm{Verify}}\left(\xi_{j,i}, \underline{\xi}_{j,i}, \bar{k}_{j,i}^{\langle c_{j,i}\rangle}\right) : i \in I_\mathrm{A} \tag{89}$$

For $i \in I_B$ and $c \in \{0, 1\}$:

$$\mu_i' *_A (\beta_{j,i,c})_A^2 =^? \nu'_{j,i,c} \wedge h_A(\beta_{j,i,c}) =^? 0 \tag{90}$$

For $i \in \mathrm{O_B}$ and $c \in \{0, 1\}$:

$$\nu'_{j,i,c} *_A (\beta_{j,i,c})_A^2 =^? \mu_i' \wedge h_A(\beta_{j,i,c}) =^? 0 \tag{91}$$

7. EVALUATE. $\mathrm{P_B}$: (let $J_{\mathrm{Ignore}} = \oslash$) (92)

Evaluate GCs. For $j \in J_E$:

$$\nu_{j,i,b_i}^{[b_i]} = \mu_i^{[b_i]} *_A \beta_{j,i,b_i} : i \in I_B \tag{93}$$

$$\xi_{j,i} \equiv k_{j,i}^{[b_i]} = \mathcal{W}_{j,i}\left(\nu_{j,i,b_i}^{[b_i]}\right) : i \in I_B \tag{94}$$

$$\mathrm{InKeys}_j^{(1)} \equiv \langle (i, \xi_{j,i}) : i \in I_{A,B}\rangle \tag{95}$$

$$\langle (i, \xi_{j,i}) : i \in \mathrm{O_B}\rangle = \mathrm{GC_{Eval}}\left(GC_j, \mathrm{InKeys}_j^{(1)}\right) \tag{96}$$

Get output encodings. For $j \in J_E$ and $i \in \mathrm{O_B}$:

$$c_{j,i} = \epsilon(\xi_{j,i}); \quad v_{j,i} = \mathcal{W}_{j,i,c_{j,i}}(\xi_{j,i}) \tag{97}$$

$$\text{If } (v_{j,i})_A^2 =^? \nu'_{j,i,c_{j,i}} \wedge h_A(v_{j,i}) =^? c_{j,i} \tag{98}$$

$$\text{then } u_{j,i} = v_{j,i} *_A \beta_{j,i,c_{j,i}} \tag{99}$$

$$\text{else } J_{\mathrm{Ignore}} = J_{\mathrm{Ignore}} \cup \{j\} \tag{100}$$

$$\text{For } i \in \mathrm{O_B} : z_i = \cup_{j \in J_E \setminus J_{\mathrm{Ignore}}} (c_{j,i}, u_{j,i}) \tag{101}$$

$$\text{If } \max\{\#(z_i) : i \in \mathrm{O_B}\} =^? 1 : \tag{102}$$

$$\text{then } \left(b_i, \mu_i^{(b_i)}\right) \leftarrow z_i : i \in \mathrm{O_B} \tag{103}$$

else enter Forge-and-Lose path: (104)

$$t_A = \mathrm{Get_{Trapdoor}}(z_{\mathrm{O_B}}) \tag{105}$$

$$b_i = \mathrm{Get_{Bit}}[t_A](\mathcal{U}_i') : i \in I_\mathrm{A} \tag{106}$$

$$\langle (i, b_i) : i \in \mathrm{O_B}\rangle = C(\langle (i, b_i) : i \in I_{A,B}\rangle) \tag{107}$$

$$(u_i : h(u_i) = b_i) \leftarrow z_i : i \in \mathrm{O_B} \tag{108}$$

8. APPLY BITCOM PERMUTATIONS. For $p \in \{A, B\}$:

$$\mathrm{P}_p : \sigma_i^{(b_i)} = \mu_i^{(b_i)} *_{\bar{p}} \gamma_i : i \in I_p \cup \mathrm{O}_p \tag{109}$$

$$\mathrm{P_A}, \mathrm{P_B} : \sigma_i' = \mu_i' *_{\bar{p}} (\gamma_i)_{\bar{p}}^2 : i \in I_p \cup \mathrm{O}_p \tag{110}$$

9. FINAL OUTPUT.

$$\mathrm{P_A} \downarrow: \left\langle \left(b_i, \sigma_i^{(b_i)}, \sigma_i'\right) : i \in I_\mathrm{A}\right\rangle, \langle \sigma_i' : i \in IO_\mathrm{B}\rangle \tag{111}$$

($\mathrm{P_A}$ outputs even if $\mathrm{P_B}$ has aborted after step (71))

$$\mathrm{P_B} \downarrow: \langle \sigma_i' : i \in I_\mathrm{A}\rangle, \left\langle \left(b_i, \sigma_i^{(b_i)}, \sigma_i'\right) : i \in IO_\mathrm{B}\right\rangle \tag{112}$$

Fig. 8: **Protocol – 1-output S2PC-with-BitComs**

8. Apply BitCom Permutations
   Both parties apply the previously decided random permutations directly to the (initial) outer encodings (the decommitments related with their own input bits) (109), and apply the square of the permutations as permutations to the respective (initial) input and output BitComs of both parties (110). The resulting permuted values are dubbed final encodings and final UH-BitComs, respectively.
9. Final Output
   Each party outputs locally the final acpBC of the bits of the other party, and the bits, the final bit encodings and respective final BitComs of their own bits ((111) and (112)). If, as part of a larger protocol, P_B needs to interact with P_A after learning the private input, then P_B takes measures (e.g., a time delay) to ensure that P_A does not find the path via which the output of P_B was obtained (normal vs. forge-and-lose).

D Random BitCom permutations

In the early Produce initial BitComs stage of the overall S2PC-with-BitComs protocol, both parties decide (initial) BitCom values related with the circuit input and output bits. In particular, P_A decides initial UH BitComs for her own input bits and for the output bits of P_B, while P_B decides initial UH BitComs for his own input bits. However, in order to emulate what a trusted third party would choose, the protocol needs to ensure that the final UH BitCom values of the protocol are random. This is achieved by applying a random permutation that leads the initial values (possibly non-random) into the final values (random for sure). In practice, this can be achieved with the parties performing a coin-tossing to decide, for each input and output wire, a random encoding of bit 0 (within the respective 2-to-1 square scheme). By means of group multiplication, this random encoding permutes the decommitments (themselves being bit encodings), and its square permutes the respective UH BitCom (squares of the respective bit encodings). Given the XOR-homomorphic properties, the final committed bit in each wire is preserved (it is the result of summing 0 to the initial bit), and the known decommitment and respective BitCom remain consistent among themselves (i.e., the former is a proper square-root of the later, encoding the correct bit). Overall, one group element needs to be decided per input and output bit of each party, independently of the number of GCs.

Combining two contributions. In order to withstand a possibly malicious party, each random group element being coin-tossed needs to be decided as a combination of two *contributions*, one from each party, so that if at least one contribution is uniformly random (from a honest party) then the final combined value is uniformly random for sure. If each party proposes her contribution as a group element, if the two contributions are independent, and if at least one of the contributions is randomly uniform from the group set, then the combination of contributions by means of the group operation yields a uniformly random group-element. This is a good approach for the purpose of the intended coin-tossing, because the group elements in class 0 form a group under the multiplication operation of the 2-to-1 square scheme. Furthermore, since the intended coin-tossing requires deciding several group elements from two groups, each party can instead propose its contribution as a vector of group elements in the respective groups (since the *direct product* of groups is itself a group, the coin tossing can be described as if it were to decide a single group element).

Full simulability. Some sophistication is needed to ensure that the overall S2PC-with-BitComs protocol is secure within the real/ideal simulation paradigm. Specifically, since the final permuted UH-BitCom values are explicit part of the output of the overall protocol, this coin-tossing of permutations needs to be *fully-simulatable*. First, the probability of abort needs to be identical (or indistinguishable) in the real and ideal worlds. Second, a simulator with black-box rewindable access to any real malicious party should be able to lead the final outcome to be whatever the trusted party in the ideal world has decided, with the probabilistic exception of abort. It is worth emphasizing that full-simulability is a requirement stronger than needed in the coin-tossings used to decide the C&C challenge configuration and in the ZKPoKs.

A fully-simulatable coin-tossing protocol. In order to have the ability, in the proof of security, to refer to some intermediate stages of the coin-tossing (as a sub-protocol), this paragraph describes a specific fully-simulatable coin-tossing protocol, namely a *parallel coin-tossing*[19] protocol proposed by Lindell [Lin03] (with some slight adjustments, to consider coin-tossing of a group element instead of a bit-string). Figure 9 gives a description, using succinct notation. The two parties are named P_1 and P_2, respectively being the first and second party to learn the outcome of the coin-tossing (assuming there is no abort). The goal is for the parties to decide a uniformly random element from a group set, assuming the respective group operation is efficient to compute (113). The parties agree on a probabilistic UB commitment scheme specification (114) that allows efficient ZKPs as needed in the remainder of the protocol. The coin-tossing starts with P_1 selecting her contribution as a random group element (115), committing to it using the UB commitment scheme (116 and 117) and giving a *ZKA of knowledge* (ZKAoK) of the respective decommitment to P_2 (118). P_2 then selects his own contribution as a random group element and sends it in the clear to P_1 (119). P_1 then computes the final permutation as the group multiplication of the two group elements and sends the result to P_2 (120). At this point, P_2 still needs to get some assurance that the outcome is correct, namely that there exists a contribution (from P_1) that is consistent with the initial commitment (121) and would lead to the final outcome (122). P_1 gives a *zero-knowledge argument* (ZKA) of such predicate (123).

Note: in the proof of security, simulation requires using the rewinding capability to convince a malicious P_2 of an incorrect statement, namely that the committed contribution is consistent with the final result (when in fact the simulator manipulates the final result independently of the initially committed contribution).

An instantiation. The protocol can be instantiated based on El-Gamal encryption [ElG85] and the *decision Diffie-Hellman* assumption (suggestion from [HL10]). The UB commitment of the contribution of P_1 is made by means of an El-Gamal encryption. The ZKAoK consists on *proving* knowledge of a discrete-log (the secret encryption key), e.g., based on a ZK adaptation of Schnorr's protocol [Sch91, § 2]. The last ZKA corresponds to *proving* that some tuples of four elements are Diffie-Hellman tuples, i.e., that two group-elements have the same discrete-log base two respective generators. A ZK adaptation of the Chaum-Pedersen proof of knowledge of the common discrete log underlying a Diffie-Hellman tuple can be used [CP93, § 3.2]). A ZK adaptation of both sub-protocols can be found in [LPS08, appendix A], using notation related to *elliptic curve cryptography*.

[19] The adjective *parallel* refers to the element being decided having up to a polynomial number of bits, even though the description that follows focuses on a *single* group element.

Common input of P_1 and P_2:

$$(\mathbb{G}, *) \text{ (group set and group operation)} \tag{113}$$

$$\mathscr{C}_{\mathrm{UB}} \text{ (specification of UB commitment scheme)} \tag{114}$$

Commit contribution by P_1.

$$\mathrm{P}_1 : \gamma^{(1)} \leftarrow^{\$} \mathbb{G} \tag{115}$$

$$\mathrm{P}_1 : \left\langle \underline{\gamma}^{(1)}, \overline{\gamma}^{(1)} \right\rangle \leftarrow^{\$} \mathscr{C}_{\mathrm{UB}}[\gamma^{(1)}] \tag{116}$$

$$\mathrm{P}_1 \rightarrow \mathrm{P}_2 : \overline{\gamma}^{(1)} \tag{117}$$

ZK argument of knowledge. $\mathrm{P}_1 \leftrightarrow \mathrm{P}_2$:

$$\mathrm{P}_1(\gamma^{(1)}, \underline{\gamma}^{(1)}) \leftrightarrow \mathrm{P}_2 : \mathrm{ZKAoK}_{\mathscr{C}^{-1}}(\overline{\gamma}^{(1)}, \mathscr{C}_{\mathrm{UB}}) \tag{118}$$

(P_1 as prover (P); P_2 as verifier (V))

Contribution by P_2.

$$\mathrm{P}_2 \rightarrow \mathrm{P}_1 : \gamma^{(2)} \leftarrow^{\$} \mathbb{G} \tag{119}$$

Reveal final random group element.

$$\mathrm{P}_1 \rightarrow \mathrm{P}_2 : \gamma = \gamma^{(1)} * \gamma^{(2)} \tag{120}$$

ZK argument. $\mathrm{P}_1 \leftrightarrow \mathrm{P}_2$:

$$\mathrm{Pred} \equiv \left(\exists \gamma^{(1)}, \underline{\gamma}^{(1)} : \mathrm{Pred}_1 \wedge \mathrm{Pred}_2 \right)$$

$$\mathrm{Pred}_1 \equiv \mathscr{C}_{\mathrm{Verify}} \left(\mathscr{C}_{\mathrm{UB}}, \gamma^{(1)}, \underline{\gamma}^{(1)}, \overline{\gamma}^{(1)} \right) \tag{121}$$

$$\mathrm{Pred}_2 \equiv \gamma =^{?} \gamma^{(1)} * \gamma^{(2)} \tag{122}$$

$$\mathrm{P}_1 \left(\gamma^{(1)}, \underline{\gamma}^{(1)} \right) \leftrightarrow \mathrm{P}_2 : \mathrm{ZKA}_{\mathrm{Pred}} \left(\gamma, \overline{\gamma}^{(1)}, \gamma^{(2)}, \mathscr{C}_{\mathrm{UB}} \right) \tag{123}$$

(P_1 as prover (P); P_2 as verifier (V))

Fig. 9: **Protocol – fully simulatable coin tossing of a group element.** The protocol is adapted from [Lin03], by replacing bit-strings and $\oplus$ with a generic group elements and a respective group operation, and simplified by not considering (in (120)) a function applied to the combination of the two contributions, and further simplified by leaving implicit the output and the actions in case of abort. **Legend:** $\gamma^{(p)}$ (contribution by P_p); $\overline{\gamma}$ (commitment of contribution); $\underline{\gamma}$ (private value required for decommitment of contribution).

Partitioning the commitments. The size of group elements associated with the El-Gamal scheme needs to be at least as high as the number of bits being committed. However, the overall S2PC-with-BitComs protocol requires the decision of many bits, namely in quantity sufficient to encode several group elements (from within the 2-to-1 square scheme), each of them of a possibly large size. Thus, the commitment can be partitioned into a tuple of smaller commitments, such that the El-Gamal parameters can be small. For example, a choice of 3,072-bit Blum integers for the UH BitComs of the global S2PC-with-BitComs protocol would correspond to a size 12 times higher than the 256-bit group-elements (based on *elliptic curve cryptography*) that could be used for the El-Gamal scheme. In such case, the number of commitments needs to be approximately (slightly larger than) 12 times the overall number of input and output wires, in order to obtain enough random bits.

Complexity. The cost of the protocol, in terms of number of communicated group elements and exponentiations (using the El-Gamal parameters) is now summarized (upon inspection of [LPS08, appendix A] and [LP11, app. B.1]).

The cost of a ZKAoK of discrete log is 7 communicated group elements and 4 exponentiations by P_1 and 5 exponentiations by P_2. By using the same El-Gamal encryption key across the different commitments, the ZKAoK of discrete log only needs to be performed once, thus having its cost amortized. The main cost of the coin-tossing comes from the ZKA of Diffie-Hellman tuple and the exchange of commitments and group elements.

– **Number of communicated group elements:** 8 for the ZKAoK of Diffie-Hellman tuple; and 4 for the committing, revealing and sending of the contributions. The communication of one group element (the second generator used by P_2 to make an UH commitment of the challenge) in each ZKA of Diffie-Hellman tuple can be eliminated by using the element selected by P_1 in the initial ZKAoK of discrete log. Thus, the overall communication complexity requires about 11 group elements per coin-tossing. For a conservative measure, 12 group elements can be still considered as an upper bound (e.g., accounting with the amortized cost from the ZKAoK of DL).

– **Number of exponentiations:** 5 by P_1 and 7 by P_2, for the ZKA of Diffie-Hellman; 2 by P_1, for the commitment. Some double exponentiations can be optimized to require less computation than two individual exponentiations (observation from [LP11, app. B.1]).

E Optimizations and complexity

This section considers some optimizations and analyzes the communication complexity of the S2PC-with-BitComs protocol. §E.1 describes how to apply a *random seed checking* technique [GMS08], such that the communication of some GCs and connectors can be replaced by the communication of only a few short commitments and random seeds. §E.2 describes an optimization whereby the multipliers and inner encodings used for connectors of P_A can be shortened in size, taking advantage of the unconditional hiding property of the respective 2-to-1 square scheme. §E.3 gives a detailed description of the optimized protocol. §E.4 makes a concrete analysis of the communication complexity of different components of the protocol (Table 3), comparing the effect of different optimizations, and comparing two concrete examples of (1-output) secure evaluation of circuits with different proportions of number of input and output wires vs. number of multiplicative gates, and comparing the effect of different C&C methods and specific instantiations of security parameters (Tables 4 and 5).

E.1 Random Seed Checking

As pointed out by Goyal et al. [GMS08], the communication complexity associated with the transmission of GCs within a C&C approach can be significantly improved by implementing a *random seed checking* (RSC) technique. The technique can be derived from two simple observations. First, the sending of GCs in the COMMIT stage can be deferred to the RESPOND stage, as long as P_A commits to them during the COMMIT stage. Second, in a typical C&C, the VERIFY stage only requires P_B to know elements that do not depend on the private inputs of P_A and P_B, and so can be generated from a short random seed and other public information. Based on these observations, the technique is implemented as follows. In the COMMIT stage, for each challenge index, P_A generates the necessary elements based on a small random seed and then sends only a short (compressive) commitment (at least computationally biding) of the generated elements. Henceforth, these are denoted as RSC commitments, as differentiation from other commitments used within the protocol. Then, in the RESPOND stage: for each *verification* challenge, P_A only sends the respective small random seed, thus allowing P_B to make all respective verifications; for each *evaluation* challenge, P_A simply sends the generated elements (but not the random seed) and the respective responses for evaluation. The optimization in communication is clear: ignoring the size of the short RSC commitments and random seeds, the communication associated with verification challenges is eliminated,[20] while the one associated with evaluation challenges remains the same.

The RSC technique can also be integrated into the new S2PC-with-BitComs protocol. Furthermore, the technique can be adapted to also encompass the communication of *connectors*. For verification challenges it is possible to completely eliminate the communication of group elements. For evaluation challenges it is possible to reduce the communication of group elements, by avoiding those which are still related with the VERIFY stage.

[20] The RSC commitments could be replaced by a single one, but the respective gain would be irrelevant.

The detailed description is given in §E.3 and Fig. 10. The following paragraph gives a rough sketch description, in order to convey some initial intuition (assuming familiarity with the non-optimized protocol description).

– **Input wires of $\mathbf{P_A}$.**
  - COMMIT stage: the permutation bit and respective encoding (the *multiplier*) are pseudo-randomly generated, but the RSC commitment is only based on the square of the *inner encoding* (i.e., the square of the encoding of the permuted bit).
  - *Verification* challenges: $\mathrm{P_B}$ regenerates the *multiplier* and then uses the homomorphic property to obtain the square of the *inner encoding*, which is then used to verify consistency against the RSC commitment.
  - *Evaluation* challenges: $\mathrm{P_A}$ reveals the *inner encoding*, the respective circuit input key and the commitment of the complementary key. These elements also allow $\mathrm{P_B}$ to confirm consistency against the RSC commitment, but without $\mathrm{P_A}$ having to reveal the known permutation bit, the respective encoding (ie., the *multiplier*), or the key corresponding to the complementary bit.
  - Number of group-elements: for each evaluation challenge, only 1 group element (the *inner encoding*) needs to be sent for each input wire of $\mathrm{P_A}$, along with the revealing of an input key and one commitment of an input key.

– **Input wires of $\mathbf{P_B}$.**
  - COMMIT stage: the inner encodings (of 0 and 1) are pseudo-randomly generated, and the RSC commitment is based on their squares.
  - *Verification* challenges: $\mathrm{P_B}$ regenerates the inner encodings and their squares and verifies consistency against the RSC commitment.
  - *Evaluation* challenges: $\mathrm{P_A}$ reveals the two multipliers that lead the outer encoding (the decommitment of the input BitCom) into the respective inner encodings; even though $\mathrm{P_B}$ only knows one outer encoding, it can use the homomorphic properties to obtain, from the two multipliers and from the outer square (the input UH BitCom), the two inner squares, and thus verify that they are consistent with the RSC commitment.
  - Number of group-elements: for each evaluation challenge, only two group elements (the multipliers) need to be sent for each input wire of $\mathrm{P_B}$.

– **Output wires of $\mathbf{P_B}$.** The procedure is symmetric to the case of input wires of $\mathrm{P_A}$, with only two multipliers being sent for each output wire of $\mathrm{P_B}$, for each evaluation challenge.

E.2 Shorter UH BitComs of input bits of $\mathbf{P_A}$

In the context of reusability of BitComs, it is important that parties remain *bound* to the respective UH BitComs in the long term; i.e., that even in the long term they do not become able to decommit a value different from the one that had been initially committed. This is specially relevant if the circuit input or output bits are supposed to be linked to other executions in the future. However, some security properties of other components of the protocol only need to hold during the protocol execution.

Following the above observation, the protocol can be optimized in communication complexity by reducing the size of some group elements associated with connectors associated with input wires of $\mathrm{P_A}$. For these wires, the BitCom of a permuted bit (i.e., the square of an *inner encoding*), which varies with the wire and with the GC, is used only to bind $\mathrm{P_A}$ to the circuit input key that needs to be disclosed in an evaluation challenge, or to the *multiplier* that needs to disclosed in a verification challenge. However, the pair of circuit input keys are independent of the values of the group elements, i.e., the group elements do not reveal any information about the pair of circuit input keys. This means that the binding property of the UH BitComs associated with these elements only needs to hold during the execution of the protocol. Thus, an optimization is possible by having $\mathrm{P_A}$ *copying* her input bits from the initial (long) UH BitComs into new (shorter) UH BitComs, and then using the shorter BitComs to substantiate the connectors. This *copying* operation can be achieved with a standard and efficient ZKP and then the remaining protocol proceeds seamlessly, as if using the original BitCom scheme.

For example, if it is adequate to define a time-out duration of 1 minute for an oblivious AES-128 evaluation (e.g., see some benchmarks taking about 1 second to execute [KSS12]), then it is valid to use shorter Blum integers assumed not to be factorable in such amount of time. For example, one could propose to use original (long) BitComs with 3,072 bits in size for BitComs of the input bits, and new (short) BitComs with 1024 bits for use for the connectors of $\mathrm{P_A}$.[21] In this case, the communication related with a ZKP of equivalence between *short* and *long* BitComs is approximately 67% of the communication of the ZKPoK of equivalence between UB and UH Bitcoms – this is somewhat irrelevant in the overall protocol, because it does not depend on the number of GCs. On the other hand, the size of exchanged group elements related with connectors of input of $\mathrm{P_A}$ is reduced to about 33% – this is relevant in the overall size occupied by the connectors.

The technique as described cannot be applied to connectors of $\mathrm{P_B}$, because the respective group elements (namely the inner encodings) actually determine the circuit-input keys (possibly with the help of widgets). In particular, if after the protocol $\mathrm{P_B}$ would break the commitment of the inner encodings (i.e., find square-roots of the inner squares), then it would be able to find all the circuit input keys of $\mathrm{P_B}$ for the evaluation GCs. This would constitute a privacy breach.

E.3 Description of optimized version

The optimized version of the protocol is described in Fig. 10 using succinct notation. For simplicity and optimization (avoiding widgets), the garbling scheme is assumed to allow the circuit input and output keys to be specified prior to the construction of GCs (this is consistent with standard garbling scheme proposals).

0. Setup. This stage follows (124) as in the unoptimized case, and additionally it contains the agreement of an additional (shorter) security parameter related with the "short UH-BitComs of $\mathrm{P_A}$" optimization (125).
1. Produce initial BitComs. The stage begins by executing the full stage of the non-optimized protocol (126), except for not doing the ZKPoK of equivalent decommitments between the (long) UH-BitCom scheme and the UB-BitCom scheme used by $\mathrm{P_A}$ to commit her input bits. Then, the stage continues with $\mathrm{P_B}$ generating a shorter 2-to-1 square scheme (127), sending its public part to $\mathrm{P_A}$ (128), and giving a ZKP of correctness that also includes a ZKPoK of trapdoor (for simplicity the succinct notation is based on the instantiation of Blum integers) (129).[22] Then, $\mathrm{P_A}$ produces respective short encodings to her input bits (130) and sends the respective

[21] This paragraph merely intends to exemplify the effects that a smaller modulus for the connectors of $\mathrm{P_A}$ may have in the communication complexity of the protocol, but not to discuss what are the shorter adequate sizes. (See [KAF$^+$10] for a report on the factorization of a RSA integer with 768 bits.)

[22] It is important that this scheme is new to $\mathrm{P_A}$, so that $\mathrm{P_A}$ cannot try to break it prior to the protocol execution.

short BitComs (i.e., squares) to P_B (131). Finally, P_A gives a ZKPoK of equivalent decommitments between the three commitment schemes with which the input bits have been commited, namely the long UH-BitCom scheme, the short UH-BitCom scheme, and the (also long) UB-BitCom scheme (132). This can be performed with the efficient ZKPoK of equivalent decommitments described in Fig. 15. For simplicity of description, henceforth the variable names are swapped between the long and the short UH-BitComs schemes (133), so that the continuing description is aligned with the variable names used in the non-optimized protocol.

2. Commit. For each of the s challenge indices, P_A selects a random seed (134) and uses it as input in a secure pseudo-random generator that determines all probabilistic choices necessary in the procedure that follows.
   - For each input wire of P_A: P_A generates a pseudo-random permutation bit (135), and a respective pseduo-random encoding (a group element, dubbed *multiplier*) (136); then, P_A multiplies the initial BitCom and the square of the multiplier, in order to obtain a square (dubbed *inner square*) that serves as BitCom of the permuted bit (137); P_A also generates two pseudo-random circuit input keys (138), and two pseudo-random elements necessary to generate the two commitments of the respective circuit input keys (139); then, P_A deterministically computes the respective two key commitments (140); finally, P_A determines a permutation of the pair of commitments, according to the permutation bit (141 and 142).
   - For each input wire of P_B: P_A generates two pseudo-random independent group elements, dubbed *inner encodings*, respectively encoding bits 0 and 1 (143); from each *inner encoding* alone (i.e., not depending explicitly on the original random seed), P_A uses a deterministic procedure to decide a respective circuit input key (e.g., extracting the key from a pseudo-random generator that uses the inner encoding as seed) (144); P_A also computes the square of the inner encodings (145).
   - P_A combines a sequence of all generated circuit input keys (146) and uses them to generate a GC which accepts those keys in the respective input wires (147). As part of building the GC, P_A also obtains the respective output keys (148).
   - For each circuit output wire of the GC, P_A uses a pseudo-random generation procedure to produce, from each output key alone, one respective *inner encoding* (group element encoding the bit underlying the respective output key) (149). In this optimized version it is assumed that widgets are not necessary. Then, P_A also sompute the respective inner squares (150).
   - P_A aggregates all the information that would have been sent in the unoptimized version of the protocol. Specifically, P_A aggregates the permuted pairs of commitments of circuit input keys of P_A (151), the inner squares related with input wires of P_A (152), the inner squares related with input and output wires of P_B (153), and the GC into a tuple of information (154). This tuple congregates all information that needs to be committed for this challenge index. Since it does not depend on any private information of P_A other than the random seed, it can be fully regenerated by P_B once P_B receives the random seed.
   - Finally, P_A computes a compressive short commitment of the tuple, dubbed *RSC commitment* (155), and sends it to P_B (156).[23]

[23] The compressiveness of the overall RSC commitment is necessary to make worth the RSC technique altogether. For example, a practical size of output may be 256 bits, in order to have 128 bits of security under birthday attacks. The extra randomness needed to verify the commitment is avoidable in practical commitment schemes, because in this case the value being committed already has enough unpredictability, but was left in the succinct notation for the sake of generality.

It is worth noticing that, in comparison with the unoptimized protocol (§C), the generation of connectors of input and output wires of P_B is different: the inner encodings are generated first, and then the multipliers are determined from them.

3. Challenge. This stage follows (157) as in the unoptimized case.
4. Decide UH-BitCom Permutations. This stage follows (158) as in the unoptimized case.
5. Respond.
   - For each verification index, P_A simply reveals the respective random RSC seed (159) and (if necessary) the randomness required to verify the respective RSC commitment (160).
   - For each evaluation index, P_A sends the randomness required to verify the respective RSC commitment (161), and, instead of also sending the RSC seed, sends the GC (162) and the following elements:
     - for each input wire of P_A: the permuted bit (163); the respective inner encoding (164); the circuit input key corresponding to the input bit of P_A and the randomness used to generate the respective key commitment (165); and the commitment of the circuit input key corresponding to the complementary bit (166);
     - for each input wire of P_B: the multipliers that lead the outer encodings (the decommitments of the input UH-BitComs) into the inner encodings (167).
     - for each input wire of P_B: the multipliers that lead the inner encodings into the outer encodings (the decommitments of the output UH-BitComs) (168).
6. Verify.
   - For each verification index, P_B uses the respective RSC seed to recompute locally all the elements that were used to prepare the RSC commitment (see steps (135) through (154)) (169). Specifically, P_B builds the GC, the permuted pairs of commitments of input keys of P_A, the inner squares for wires of P_A and P_B, then combines them into a tuple and combines them into a tuple. P_B then verifies that the obtained tuple is consistent with the respective RSC commitment (170).
   - For each evaluation index, P_B also needs to regenerate the RSC commitment, but now without using the random RSC seed. Instead, P_A uses the other elements received from P_A, as follows, for each *evaluation* index:
     - **Congregate input keys of P_A.** For each input wire of P_A, P_B computes the commitment of the received input key (also using the received associated randomness) (171). Then, P_B congregates the commitments of all the input keys (the ones just computed, and the complementary ones directly received (166)) (172).
     - **Congregate input *inner squares* of P_A.** P_B computes the square of the received inner encodings (of the permuted bit) (173) and congregates them all (174);
     - **Congregate input and outut *inner squares* of P_B.** For each input wire of P_B, P_B computes the squares of the inner encodings, by multiplying the input BitComs with the squares of the received multipliers (175); For each output wire of P_B, P_B computes the squares of the inner encodings, by multiplying the output BitComs with the squares of the inverse of the received multipliers (176); Then, P_B aggregates them all (177).
     - P_B combines a tuple of aggregates, as a honest P_A would (178), and then verifies that it is consistent with the received RSC commitment and respective verifying randomness (179).

– Finally, $\mathrm{P_B}$ also verifies that the multipliers received for input and output wires of $\mathrm{P_B}$ are encodings of 0 (180) (without this verification, $\mathrm{P_A}$ could easily cheat by sending non-trivially correlated elements; i.e., proper square-roots encoding 1).

0. Setup.Steps (44)-(46) (see Fig. 8). (124)

$\mathrm{P_A} \leftrightarrow \mathrm{P_B} : 1^{\kappa'} : \kappa' < \kappa$ (shorter security parameter) (125)

1. Produce initial BitComs.

Same as in Fig. 8, except $\mathrm{ZKPoK}_{\mathrm{Equiv}}(\mu'_{I_\mathrm{A}}, \mathcal{U}'_{I_\mathrm{A}})$ (126)
(steps (47)-(56), except (54))

Transfer bits of $\mathbf{P_A}$ to short UH BitComs.

$\mathrm{P_B} : \langle \mathbb{G}_{B'}, h_{B'}, t_{B'} \rangle \xleftarrow{\$} \mathrm{Gen}(\kappa')$ (127)

$\mathrm{P_B} \to \mathrm{P_A} : \langle \mathbb{G}_{B'}, h_{B'} \rangle$ (128)

$\mathrm{P_B} \leftrightarrow \mathrm{P_A}(t_{B'}) : \mathrm{ZKPoK}_{\text{BI-trap}}(\mathbb{G}_{B'})$ (Fig. 17) (129)

$\mathrm{P_A} : \eta_i \equiv \eta_i^{(b_i)} \xleftarrow{\$} h_{B'}^{-1}(b_i) : i \in I_\mathrm{A}$ (130)

$\mathrm{P_A} : \eta'_i = (\eta_i)^2_{B'} : i \in I_\mathrm{A}$ (131)

$\mathrm{P_A}(\mu_{I_\mathrm{A}}, \eta_{I_\mathrm{A}}, \mathcal{U}_{I_\mathrm{A}}) \leftrightarrow \mathrm{P_B}$:
$\mathrm{ZKPoK}_{\mathrm{Equiv}}(\mu'_{I_\mathrm{A}}, \eta'_{I_\mathrm{A}}, \mathcal{U}'_{I_\mathrm{A}})$ (see Fig. 15) (132)

Swap variable names:

$``(\mathbb{G}_B, h_B, \mu_i)'' \leftrightarrow ``(\mathbb{G}_{B'}, h_{B'}, \eta_i)''$ (133)

2. Commit. $\mathrm{P_A}$: For $j \in \{1, ..., s\}$:

Select seeds. $\lambda_j \xleftarrow{\$} \mathrm{SEEDS}_\kappa$ (RSC seed) (134)

Connectors input $\mathbf{P_A}$. For $i \in I_\mathrm{A}$:

$\pi_{j,i} = \mathrm{Gen}_{\mathrm{Bit}}[\lambda_j, i]$ (135)

$\alpha_{j,i} \equiv \alpha_{j,i}^{(\pi_{j,i})} = \mathrm{Gen}_{\mathrm{Encoding}}[\lambda_j, i](\mathbb{G}_B, \pi_{j,i})$ (136)

$\nu'_{j,i} = \mu'_i *_B (\alpha_{j,i})^2_B$ (137)

$k_{j,i}^{[c]} = \mathrm{Gen}_{\mathrm{InKey}}[\lambda_j, i, c] : c \in \{0,1\}$ (138)

$\underline{k}_{j,i}^{[c]} = \mathrm{Gen}_{\mathrm{RandForCommit}}[\lambda_j, i, c]$ (139)

$\overline{k}_{j,i}^{[c]} = \mathscr{C}_{\mathrm{Commit}}[\underline{k}_{j,i}^{[c]}]\left(k_{j,i}^{[c]}\right) : c \in \{0,1\}$ (140)

$\langle c \rangle = [c \oplus \pi_{j,i}] : c \in \{0,1\}$ (141)

$\overline{\mathrm{InKeys}}_{A,j,i}^{(2)} = \left\langle \overline{k}_{j,i}^{\langle 0 \rangle}, \overline{k}_{j,i}^{\langle 1 \rangle} \right\rangle$ (142)

Connectors input $\mathbf{P_B}$. For $i \in I_B$ and $c \in \{0,1\}$:

$\nu_{j,i,c}^{[c]} \equiv \nu_{j,i,c}^{(c)} = \mathrm{Gen}_{\mathrm{Encoding}}[\lambda_j, i](\mathbb{G}_A, c)$ (143)

$k_{j,i}^{[c]} = \mathrm{Gen}_{\mathrm{InKey}}\left[\nu_{j,i,c}^{(c)}\right]$ (144)

$\nu'_{j,i,c} = (\nu_{j,i,c})^2_A$ (145)

GCs.

$\mathrm{InKeys}_j^{(2)} \equiv \left\langle \left(i, c, k_{j,i}^{[c]}\right), i \in I_{A,B}, c \in \{0,1\} \right\rangle$ (146)

$\left\langle GC_j, \mathrm{OutKeys}_j^{(2)} \right\rangle = \mathrm{Gen}_{\mathrm{GC}}[\lambda_j](C, \mathrm{InKeys}_j^{(2)})$ (147)

$\mathrm{OutKeys}_j \equiv \left\langle \left(i, c, k_{j,i}^{[c]}\right) : i \in \mathrm{O_B}, c \in \{0,1\} \right\rangle$ (148)

Connectors output $\mathbf{P_B}$. For $i \in \mathrm{O_B}$ and $c \in \{0,1\}$:

$\nu_{j,i,c} \equiv \nu_{j,i,c}^{(c)} = \mathrm{Gen}_{\mathrm{Encoding}}\left[k_{j,i}^{[c]}\right](\mathbb{G}_A, c)$ (149)

$\nu'_{j,i,c} \equiv (\nu_{j,i,c})^2_A$ (150)

Prepare RSC Commitment.

$\overline{\mathrm{InKeys}}_{A,j}^{(2)} = \left\langle \left(i, \overline{\mathrm{InKeys}}_{A,j,i}^{(2)}\right) : i \in I_\mathrm{A} \right\rangle$ (151)

$N'_{A,j} = \langle (i, \nu'_{j,i}) : i \in I_\mathrm{A} \rangle$ (152)

$N'_{B,j} = \langle (i, c, \nu'_{j,i,c}) : i \in I_B \cup \mathrm{O_B}, c \in \{0,1\} \rangle$ (153)

$\mathit{\Lambda}_j = \left(GC_j, \overline{\mathrm{InKeys}}_{A,j}^{(2)}, N'_{A,j}, N'_{B,j}\right)$ (154)

Finalize RSC Commitment.

$\mathrm{P_A} : (\underline{\mathit{\Lambda}}_j, \overline{\mathit{\Lambda}}_j) \xleftarrow{\$} \mathscr{C}_{\mathrm{Commit}}(\mathit{\Lambda}_j)$ (compressive) (155)

$\mathrm{P_A} \to \mathrm{P_B} : \overline{\mathit{\Lambda}}_j$ (RSC commitment) (156)

3. Challenge (same as in Fig. 8) (157)

4. Decide UH-BitCom Permutations. (as in Fig. 8) (158)

5. Respond.

***Verification* indices.** For $j \in J_V$:

$\mathrm{P_A} \to \mathrm{P_B} : \lambda_j$ (RSC seed) (159)

$\mathrm{P_A} \to \mathrm{P_B} : \underline{\mathit{\Lambda}}_j$ (randomness for verification) (160)

***Evaluation* indices.** For $j \in J_E$:

$\mathrm{P_A} \to \mathrm{P_B} : \underline{\mathit{\Lambda}}_j$ (randomness for verification) (161)

$\mathrm{P_A} \to \mathrm{P_B} : GC_j$ (162)

Input of $\mathbf{P_A}$. For $i \in I_\mathrm{A}$:

$\mathrm{P_A} \to \mathrm{P_B} : c_{j,i} = b_i \oplus \pi_{j,i}$ (163)

$\mathrm{P_A} \to \mathrm{P_B} : \nu_{j,i} \equiv \nu_{j,i}^{(c_{j,i})} \equiv u_i^{[b_i]} *_B \alpha_{j,i}$ (164)

$\mathrm{P_A} \to \mathrm{P_B} : \left(\xi_{j,i}, \underline{\xi}_{j,i}\right) = \left(k_{j,i}^{\langle c_{j,i} \rangle}, \underline{k}_{j,i}^{\langle c_{j,i} \rangle}\right)$ (165)

$\mathrm{P_A} \to \mathrm{P_B} : \overline{k}_{j,i}^{[1 \oplus b_i]} \equiv \overline{k}_{j,i}^{\langle 1 \oplus c_{j,i} \rangle}$ (166)

Input of $\mathbf{P_B}$. $\mathrm{P_A} \to \mathrm{P_B}$: For $i \in I_B$:

$\beta_{j,i,c} \equiv \beta_{j,i,c}^{(0)} = (1/\mu_i^{(c)}) *_A \nu_{j,i,c}^{(c)} : c \in \{0,1\}$ (167)

Output of $\mathbf{P_B}$. $\mathrm{P_A} \to \mathrm{P_B}$: For $i \in \mathrm{O_B}$:

$\beta_{j,i,c} \equiv \beta_{j,i,c}^{(0)} = (1/\nu_{j,i,c}^{(c)}) *_A \mu_i^{(c)} : c \in \{0,1\}$ (168)

6. Verify. $\mathrm{P_B}$:

***Verification* indices.** For $j \in J_V$:

Recompute steps (135)-(154) to obtain $\mathit{\Lambda}_j$ (169)

$\mathscr{C}_{\mathrm{Verify}}(\mathit{\Lambda}_j, \underline{\mathit{\Lambda}}_j \overline{\mathit{\Lambda}}_j)$ (170)

***Evaluation* indices.** For $j \in J_E$:

$\overline{k}_{j,i}^{\langle c_{j,i} \rangle} = \mathscr{C}_{\mathrm{Commit}}\left(\xi_{j,i}, \underline{\xi}_{j,i}\right) : i \in I_\mathrm{A}$ (171)

$\overline{\mathrm{InKeys}}_{A,j}^{(2)} : ((166), (171)) \to (142) \to (151)$ (172)

$\nu'_{j,i} = (\nu_{j,i})^2_B : i \in I_\mathrm{A}$ (173)

$N'_{A,j} : ((173)) \to (152)$ (174)

$\nu'_{j,i} = \mu'_i *_A (\beta_{j,i})^2_A : i \in I_B, c \in \{0,1\}$ (175)

$\nu'_{j,i} = \mu'_i *_A ((\beta_{j,i})^{-1}_A)^2_A : i \in \mathrm{O_B}, c \in \{0,1\}$ (176)

$N'_{B,j} : ((175), (176)) \to (153)$ (177)

$\mathit{\Lambda}_j : ((162), (172), (174), (177)) \to (154)$ (178)

$\mathscr{C}_{\mathrm{Verify}}(\mathit{\Lambda}_j, \underline{\mathit{\Lambda}}_j \overline{\mathit{\Lambda}}_j)$ (179)

$h_A(\beta_{j,i,c}) \stackrel{?}{=} 0 : i \in I_B \cup \mathrm{O_B}, c \in \{0,1\}$ (180)

7. Evaluate. (same as in Fig. 8)) (181)

8. Apply BitCom Permutations. (as in Fig. 8) (182)

9. Final Output. (same as in Fig. 8) (183)

Fig. 10: **Protocol – 1-output S2PC-with-BitComs (optimized).**

Remark. The number of multipliers whose class needs to be verified is equal to twice the number of input and output wires of $\mathrm{P_B}$ multiplied by the number of GCs. A trivial verification method (180) is to individually determine the class of each multiplier. However, if *group multiplication* is, in terms of computation, significantly less expensive than *class determination* (e.g., modular multiplication being less expensive than computation of a Jacobi Symbol), then the verification task can be optimized be means of a parallel statistical verification of classes. The key observation is that if there is at least one multiplier encoding bit 1, then there is a 50% probability that the product of a random subset of these multipliers will also encode bit 1. Actually, if the empty subset is excluded, the probability is larger than 50% by a positive negligible amount. Conversely, if all multipliers encode bit 0, then the product of any subset of these multipliers will also encode bit 0. Thus, if the subset verification succeeds for a number of times equal to the statistical parameter (using a different non-empty random subset every time), then $\mathrm{P_B}$ establishes that there is an overwhelming probability that all multipliers encode bit 0. Note: this type of parallel statistical verification could also be applied to the protocol without the RSC technique, though in that case the verifications of classes of group elements associates with input wires of $\mathrm{P_A}$ would have to be separated from the verifications associated with input and output wires of $\mathrm{P_B}$.

If throughout this VERIFY stage any verification has failed, then $\mathrm{P_B}$ aborts, outputting FAIL.

7,8,9. EVALUATE, APPLY BITCOM PERMUTATIONS, FINAL OUTPUT**.** The protocol continues as in the unoptimized case ((181), (182), (183)).

E.4 Communication complexity

Table 3 analyzes the communication complexity of different components of the protocol (along different rows) and under different levels of optimization (along different columns). From left to right, first the unoptimized case (Simple) is analyzed, then the result of applying the *random seed checking* technique (RSC), and finally the result of adding the optimization related with shorter BitComs in the connectors of $\mathrm{P_A}$ ($\mathrm{BC_{short}}$). For simplicity, the contribution from small components is neglected, such as the communication associated with the coin-tossings necessary to decide the C&C challenges and the challenges in ZKPoKs. Clearly, besides GCs, most of the communication cost is due to connectors, which are in number linear with the number of GCs multiplied by the number of input and output wires. In contrast, the cost of ZKPoKs and random BitCom permutations is small, because it is either linear in the statistical parameter or in the number of input and output wires, respectively, but not to their product.

Tables 4 and 5 exemplify a more concrete analysis, estimating the communication associated with the secure evaluation of two different circuits. Table 4 considers a circuit that evaluates the AES-128 function, using $6,800$ multiplicative gates (from [Bri13]), with 128 bits of input for each party, and 128 bits of output of $\mathrm{P_B}$ to hold the enciphering of the input of $\mathrm{P_B}$ using as key the input of $\mathrm{P_A}$. Table 5 considers a circuit that evaluates the SHA-256 function, with $90,825$ multiplicative gates (from [Bri13]), with 256-bits of private input for each party, and 256-bits of output of $\mathrm{P_B}$ to hold the (SHA-256) hash of the concatenated inputs.

The following parameters are assumed: a security goal of 128 bits of cryptographic security, instantiated by Blum integers with $3,072$ bits (i.e., comparable to AES-128

Table 3: Communication complexity (analytic)

Component	Method		Unoptimized (see §C)	RSC (see §E.1 and §E.1)	RSC + $\mathrm{BC}_{\mathrm{short}}$ (see §E.2 and §E.1)	Related descriptions
GCs			$(v+e)\times\lvert C\rvert\times\lvert g\rvert$	$e\times\lvert C\rvert\times\lvert g\rvert$	==	...
BitComs			$(\ell+\ell_A)\times n$	==	==	§C
Connectors	Input of $\mathrm{P_A}$		$\ell_A(v+e)(2n)+$ $\ell_A(v)2(\lvert\mathscr{C}_c\rvert+\lvert\mathscr{C}_o\rvert)+$ $\ell_A(e)(2\lvert\mathscr{C}_c\rvert+\lvert\mathscr{C}_o\rvert)$	$\ell_A(e)n+$ $\ell_A(e)(\lvert\mathscr{C}_c\rvert+\lvert\mathscr{C}_o\rvert)$	$\ell_A(e+1)n_2+$ $\ell_A(e)(\lvert\mathscr{C}_c\rvert+\lvert\mathscr{C}_o\rvert)$	§B.1, §E.1, §E.2
	Input of $\mathrm{P_B}$		$\ell_B(4v+4e)n$	$\ell_B(2e)n$	==	§B.2, §E.1
	Output of $\mathrm{P_B}$		$\ell'_B(2v+4e)n$	$\ell'_B(2e)n$	==	§B.3, §E.1
ZKPoKs	Blum integer trapdoor	of $\mathrm{P_A}$	$s'(9n/2)$	==	==	§F.2
		of $\mathrm{P_B}$	$s'(3n)$	==	$s'(3n+3n_2)$	§F.2, §E.2
	Equivalent decommitments of $\mathrm{P_A}$		$s'(4n)$	==	$s'(4n+2n_2)$	Fig. 15, §E.2
	Decommitments of UH BitComs of $\mathrm{P_B}$		$s'(2n)$	==	==	Fig. 13
Coin-tossing to decide random BitCom permutations			$\ell(12n)$	==	==	§D

Legend: # (number of), v (# of verification indices), e (# of evaluation indices), $\lvert C\rvert$ (# multiplicative gates), $\lvert\mathscr{C}_C\rvert$ (# bits of a commitment), $\lvert\mathscr{C}_O\rvert$ (# bits to open a decommit), $\lvert g\rvert$ (# bits in a garbled gate), ℓ_A (# input wires of $\mathrm{P_A}$), ℓ_B (# input wires of $\mathrm{P_B}$), ℓ'_B (# output wires of $\mathrm{P_B}$), ℓ ($=\ell_A+\ell_B+\ell'_B$), n (# bits in normal group-elements), n_2 (# bits in short group-elements), s' (statistical parameter used in ZKPoKs), == (value equal to the left cell). For simplicity, some small contributions are ignored, namely: the communication of the coin-tossing sub-protocols used to decide challenges for the C&C and for the ZKPoKs, each of which requires only committing a single short value and revealing two short values; and the communication of random seeds and commitments used in the RSC technique. The contribution of $\ell_A\times n_2$ bits, from the new UH BitComs present in the $\mathrm{BC}_{\mathrm{short}}$ optimization, is included in the "Connectors – Input of $\mathrm{P_A}$" row. The contribution from the UB BitComs of input wires of $\mathrm{P_A}$ is included in the "BitComs" row (even though they are not part of the final output).

according to the current NIST "comparable strengths" table [BBB+12]); when applying the optimization with shorter BitComs for connectors of $\mathrm{P_A}$, a secondary cryptographic security parameter of 80 bits, instantiated with a Blum integer with 1024 bits (which only needs to remain unfactorable during the protocol execution); each garbled gate occupying 384 bits (corresponding to a list of 3 AES ciphertexts each, upon using a *garbled-row reduction* technique [PSSW09]); a commitment scheme for circuit input keys of $\mathrm{P_A}$ (for connectors-in-A) using 256 bits per commitment value, and 384 bits per opening (i.e., in the *reveal* phase).[24]

The focus of the analysis considers a statistical security goal of 40 bits, which is a common benchmark. From a communication complexity perspective, the "total size" row in Tables 4 and 5 has the most important measure. Clearly, the RSC and the $\mathrm{BC}_{\mathrm{short}}$ optimizations bring benefits. Furthermore, when applying the RSC technique, lowering the number of *evaluation* challenges at the expense of over-increasing the number of verification challenges also brings a benefit in communication. For example, to securely evaluate the exemplified AES 128 circuit, with 128 bits of cryptographic security, 40

[24] 256 bits per commitment of each 128-bit key, in order to have collision resistance up to 128 bits of security under *birthday attacks*. For a commitment with 256 bits, 384 bits of opening allows a commitment scheme to be unconditionally hiding if the elements being committed have 128 bits. For a computationally hiding scheme, the opening can conceivably be reduced to only 256 bits.

- **RSC** (optimization *random seed checking* technique, either applied only to the connectors, or applied simultaneously to the connectors and the GCs)
- $\mathbf{BC_{short}}$ (optimization using shorter BitComs for the connectors of $\mathrm{P_A}$)
- **Mb** (millions of bits, rounded to the closest decimal)
- $|C|$ (number of garbled gates in the garbled circuit, i.e., # multiplicative gates in logic circuit)
- ℓ_A**,** ℓ_B**,** ℓ'_B (number of input wires of $\mathrm{P_A}$, number of input wires of $\mathrm{P_B}$, number of output wires of $\mathrm{P_B}$, respectively)
- ℓ (total number of input and output wires per circuit, i.e., $\ell_A + \ell_B + \ell'_B$)
- s (number of challenge indices, i.e., of GCs built by $\mathrm{P_A}$, satisfying $s = v + e$)
- $(e_{\mathbf{min}}, e_{\mathbf{min}})$ (interval of variation of number of evaluation challenges)
- $(v_{\mathbf{min}}, v_{\mathbf{min}})$ (interval of possible variation of number of verification challenges)
- s' (statistical security parameter, sucth that $\mathrm{Pr_{Error}} \lessapprox 2^{-s'}$)
- κ (cryptographic security parameter)
- n (number of bits per group-element in BitComs)
- n_2 (number of bits per group-element in connectors of $\mathrm{P_A}$, if using the $\mathrm{BC_{short}}$ optimization)
- $|\mathscr{C}_C|$ (number of bits per (symmetric) commitment)
- $|\mathscr{C}_O|$ (number of bits to open a commitment)
- $|gg|$ (number of bits per garbled gate)

Note (†): For the RSC and $\mathrm{BC_{short}}$ optimizations, the sizes presented for GCs and connectors are maximal – the variable number of *evaluation* challenges may lead to a lower communication complexity. The respective "(not GCs)/GCs" overhead proportion is minimal, as the size of GCs reduces more than the size of connectors. In other words, whenever the "total size" is better (lower) than shown, the respective overhead propoertion is higher. For the unoptimized case with $v < s$ the size of GCs is constant, but the size of connectors also varies with e and v. In that case, the "(not GCs)/GCs" values are the most likely (which happens for $e = v = s/2$).
Note (‡): The contribution of the coin-tossing sub-protocols used in the ZKPoKs and to decide the C&C partition is ignored.

Fig. 11: Common legend for Tables 4 and 5

bits of statistical security and using 76 GCs out of which at most 10 are evaluated, the protocol requires about 62 million bits, i.e., about 8 Mega Bytes. The corresponding communication for the SHA-256 circuit is about 52 Mega Bytes.

It is also interesting to look at the proportional overhead brought by the elements that are not GCs. For example, in the column reporting results under the RSC (applied to connectors and GCs) and $\mathrm{BC_{short}}$ optimizations, for 128 of cryptographic security, 40 bits of statistical security, and using 41 GCs, the overhead proportions for the AES-128 and SHA-256 circuits considered are, respectively, 103% and 15%. Clearly, the overhead proportion decreases with the ratio between number of input and output wires and the number of multiplicative gates, which for the two circuits is, respectively, 5.9% and 0.8%.

Even though the overheads from non-GC elements may seem somewhat large, any overhead below 200% represents less space than what would be required by the additional GCs in C&C protocols that require a majority of evaluated GCs to be correct (because these protocols require about 3 times more GCs). Furthermore, other protocols usually also have an overhead related with ensuring the consistency of input and output across different GCs.

If parties are not willing to apply the RSC technique to the GCs, e.g., not to increase the computation related with GCs, then the estimate for the AES-128 circuit is about 165 Mega bits (see Table 4). The comparable reported overhead for the SEOC method is 177,725,440 bits [Lin13], which is of the same order of magnitude.

Future optimizations in the way connectors are committed and verified might allow further reduction of the overhead, by reducing the number or size of group elements that need to be communicated.

In the two rightmost columns, for the purpose of comparison, different instantiations of security parameters are also considered. The before-last column uses Blum integers with 1024 bits (for 80 bits of cryptographic security), which allows a further improvement in communication but is not recommendable for long term privacy. It may (with caution)

Table 4: Communication in S2PC of AES-128

Circuit	AES-128 ($\|C\| = 6,800$, $\ell_A = \ell_B = \ell'_B = 128$, $\ell/\|C\| = 5.9\%$)							
Optimization RSC	No	Yes @ Connectors		Yes @ GCs + Connectors				
Optimization BC$_{\text{short}}$	No		Yes	No	Yes		No	
C&C restriction	$v < s$	$\lceil s/2 \rceil \leq v < s$				$e \in [s'/4]$	$e \in [s'/5]$	
Security params. (κ, s')	(128,40)						(80, 40)	(128,128)
# GCs (s)	40	41				76	123	365
$(v_{\min}, v_{\max})$	$(0, 39)$	$(21, 39)$				$(66, 75)$	$(115, 122)$	$(340, 364)$
$(e_{\min}, e_{\max})$	$(1, 40)$	$(1, 20)$				$(1, 10)$	$(1, 8)$	$(1, 25)$
Group-elements (n, n_2)	(3072, -)		(3072, 1024)	(3072, -)	(3072, 1024)		(1024, -)	(3072, -)
Size symmetric primitives	$\|\mathscr{C}_C\| = 256$, $\|\mathscr{C}_O\| = 384$, $\|gg\| = 384$							
†GCs (Mb)	104.4	107.1		52.2		26.1	20.9	65.3
BitComs (Mb)	1.6						0.5	1.6
†Connectors (Mb)	147.1	41.0	35.5	41.0	35.8	18.0	5.9	44.8
ZKPoKs (Mb)	1.8		2.2	1.8	2.2		0.6	7.0
BitCom perms (Mb)	14.2						4.7	14.2
‡Total size (Mb)	269.1	165.3	160.6	110.4	105.7	62.0	32.6	132.8
†,‡(not GCs) / GCs	158%	55%	50%	112%	103%	138%	56%	103%

(See legend in Fig. 11)

be for an application where the binding to the BitComs and the privacy of the input of P_A only needs to hold in the short term. In contrast, the last column shows an example with 128 bits of statistical security, which may be too high for practical purposes.

Pipelining. If the parties *pipeline* computation and communication, namely the generation and sending of garbled gates (P_A), or the receiving and evaluation of garbled gates (P_B), the memory required by each party can be lower than the communication performed [HEKM11]. When generating a GC, P_A discards each garbled gate as soon as it uses it in the computation of the (compressive) RSC commitment (in the COMMIT stage), or as soon as it sends it to P_B (in the RESPOND stage, for evaluation GCs), and discards the keys of each intermediate wire as soon as all related garbled-gates have been generated. For each *evaluation* GC, P_B discards each garbled gate as soon as it obtains its output wire key, and discards each (intermediate) wire key as soon as all respective garbled-gates have been evaluated. For each *verification* GC, P_B simply generates the GC, from the respective random seed, as described for P_A. This technique allows a significant reduction in memory (i.e., the amount of information that needs to be stored simultaneously). As noted in [KSS12], a down-side of implementing pipelining in a C&C protocol is that P_A needs to generate twice each evaluation GC (assuming it does indeed discard the GCs from memory).

F Zero Knowledge Proofs

The described S2PC-with-BitComs protocols use several ZKPoKs related with the 2-to-1 square schemes. §F.1 describes these ZKPoKs, and also a ZKP that can be useful to link

Table 5: Communication in S2PC of SHA-256

Circuit	AES-256 ($\|C\| = 90,825$, $\ell_A = \ell_B = \ell'_B = 256$, $\ell/\|C\| = 0.8\%$)							
Optimization RSC	No	Yes @ Connectors		Yes @ GCs + Connectors				
Optimization BC_{short}	No		Yes	No	Yes		No	
C&C restriction	$v < s$	$\lceil s/2 \rceil \leq v < s$				$e \in [s'/4]$	$e \in [s'/5]$	
Security params. (κ, s')	(128,40)						(80, 40)	(128,128)
# GCs (s)	40	41				76	123	365
$(v_{\min}, v_{\max})$	$(0,39)$	$(21,39)$				$(66,75)$	$(115,122)$	$(340,364)$
$(e_{\min}, e_{\max})$	$(1,40)$	$(1,20)$				$(1,10)$	$(1,8)$	$(1,25)$
Group-elements (n, n_2)	(3072, -)		(3072, 1024)	(3072, -)	(3072, 1024)		(1024, -)	(3072, -)
Size symmetric primitives	$\|\mathscr{C}_C\| = 256$, $\|\mathscr{C}_O\| = 384$, $\|gg\| = 384$							
†GCs (Mb)	1,395.1	1429.9		697.5		348.8	279.0	871.9
BitComs (Mb)	3.1						1.0	3.1
†Connectors (Mb)	294.3	81.9	71.7	81.9	71.7	36.0	11.8	89.6
ZKPoKs (Mb)	1.8		2.2	1.8	2.2		0.6	7.0
BitCom perms (Mb)	28.3						9.4	28.3
‡Total size (Mb)	1,722.6	1,545.2	1,535.3	812.8	802.9	418.4	301.9	999.9
†,‡(not GCs) / GCs	23%	8%	7%	17%	15%	20%	8%	15%

(See legend in Fig. 11)

executions, assuming that the specification of the 2-to-1 square scheme is correct. Then, §F.2 considers a proof of correctness of a Blum integer (which is also a ZKPoK of its prime factors) which can be used as proof of correctness of the specification of a 2-to-1 square-scheme based on properties of Blum integers.

F.1 Several ZKPoKs

Each ZKPoK is described as a parallelization of several rounds of a typical Σ-protocol structure (commit-challenge-respond), in order to achieve a number of bits of statistical security equal to the number of rounds. The vector of challenges is decided via a coin-tossing sub-protocol, which does not need to be fully simulatable (as is required for the sub-protocol that decides the random BitCom permutations, described in §D), though it needs only to enable some properties to the ZKPoK:

1. When making a simulation with rewinding capable access to a black-box malicious verifier (V^*), the simulator must be able to impersonate a honest prover (i.e., one that would possess the knowledge being proven). In other words, a malicious V^* cannot distinguish the simulator from a honest prover. This is allowed by a coin-tossing sub-protocol where a simulator is able, with noticeable probability (larger than the inverse of a positive polynomial, though possibly lower than 1), to determine on its own a random vector of challenges that it wants as outcome of the coin-tossing (and assuming that there is a noticeable probability that V^* does not abort the coin-tossing execution). In the *commit* stage of the ZKPoK, the simulator first decides a random vector of challenges and prepares itself to be able, later in the *response* stage, to

give a successfully verifiable response. Then, the simulator leads the outcome of the coin-tossing to be exactly such vector of challenges.

2. When making a simulation with rewinding capable access to a black-box malicious prover (P^*) that has a noticeable probability of making a honest verifier (V) accept, the simulator (mpersonating a honest verifier) must be able to extract the elements whose knowledge is being proven. This is achieved by taking advantage of the property that the underlying private knowledge of the prover can be extracted from valid responses to several different challenges (two, in some protocols), if they are related with the same value committed by the prover in the *commit* stage. Thus, withing the coin-tossing sub-protocol, the simulator must be able to lead the outcome of the *challenge* phase to different vectors of challenges, without changing the value committed by P^*, and being able to receive the respective valid *responses*.

A possible coin-tossing protocol for determining the challenges in parallel is the same as suggested for the decision of challenges of the C&C partition (see Fig. 7), though with the following semantic adjustments:

– replacing the number of bits that need to be decided (9), either to be the number of challenges (if challenges are binary), or to be the number of challenges multiplied by the vector length of each challenge (if a challenge itself is defined as a vector of bits).
– choosing the VARIABLE mode, with any possible number of verification challenges (10), so that the final bit-string encoding the challenges is uniformly random.
– replacing P_A and P_B by Prover (P) and Verifier (V), respectively.

Since the commitment scheme can be compressive, the protocol is efficient in terms of communication even when the challenges are not binary, but rather vectors of bits, as is the case in some ZKPoK protocols. For simplicity of description, the following explanations leave implicit that there are several Σ-iterations in parallel, and that the BitComs are always XOR-homomorphic with trapdoor.

ZKPoK of non-trivial square-root of 1 (see Fig. 12).

– **Setup.** As private input, P knows the trapdoor of the implicit 2-to-1 square scheme. It is assumed that this trapdoor is represented as a non-trivial square-root of 1 (i.e., an encoding that is in class 1 and is a square-root of 1) (184).
– **Procedure.** For each challenge index, P selects a random group element of class 0 (185) and sends its square to V (186). As each challenge, the parties decide a random bit (187). As response, P sends to V a square-root encoding the challenged bit (if 0, it sends the previously selected group element; if 1 it sends the result of multiplying it by the non-trivial square-root of 1) (188) . V verifies that the square-root is correct and encodes the challenged bit (189). V accepts the proof if all responses are correctly verified (190).
– **Remark.** The described protocol requires communication of 2 group elements per round, and no modular exponentiation. The protocol can be easily adapted with a *random seed checking* technique to reduce the number of communicated group elements to just those responded for bit-challenges with value 1. In the *commit* stage, P would generate each square-root from a short random seed, then locally compute all squares, and then send to P only a short commitment of all squares. In the *respond* stage: for class 0 challenges, P just sends the random seed; for class 1 challenges, P sends only the respective square-root class 1. Finally, in the *verify* stage, V is able

to reconstruct all the squares and verify that they are consistent with the RSC commitment. Thus, overall this optimization reduces the communication to an expected half of a group element per round, beside the communication associated with short random seeds and with the coin tossing of challenges and opening one of commitment.

Private input of P: NTSQRT$_1$ (the trapdoor) (184)

Commit. For $j \in [s']$:

$P : \lambda_j \leftarrow^{\$} h^{-1}(0)$ (encoding of bit 0) (185)

$P \rightarrow V : \lambda'_j = (\lambda_j)^2$ (square) (186)

Challenge.

$P \leftrightarrow V : c \leftarrow^{\$} \{0,1\}^{s'}$ (coin-toss s' bits) (187)

Respond. For $j \in [s']$:

$P \rightarrow V : r_j = \lambda_j * (\mathrm{NTSQRT}_1)^{c_j}$ (188)

(square-root encoding bit c_j)

Verify. For $j \in [s']$:

V : If $(r_j)^2 \neq \lambda'_j \vee h(r_j) \neq c_j$, then REJECT (189)

V : ACCEPT (190)

Fig. 12: **ZKPoK of non-trivial square-root of 1 (ZKPoK$_{\mathbf{NTSQRT1}}$).** Legend: NTSQRT$_1$ (non-trivial square-root of group-identity, encoding bit 1).

ZKPoK of vector of square-roots (see Fig. 13).

- **Setup.** As common input, the parties know a vector of squares (191). As private input, P knows a respective vector of square roots (192).
- **Procedure.** For each challenge index, P selects a random group element (dubbed *mask*) (193) and commits by sending its square to V (194). As each challenge, the parties decide a random subset of the squares (encoded as a vector of bits, with the positions with value 1 representing the selected squares) (195). As response, P sends to V a square root of the masked version (i.e., multiplied by the mask) of the product of the respective subset of squares (196). V verifies that the square-root is correct (197). V accepts the proof if all responses are correctly verified (198).
- **Remarks.** Since the mask encodes a random bit in each challenge, V does not learn the XOR of the bits encoded by the square-roots associated with the selected subset of indices. This is essentially the Feige-Fiat-Shamir ZKPoK of a vector of square-roots [FFS88], enhanced only with a coin tossing protocol to decide the challenges (instead of being just selected by V).

Common input of P and V :

$\langle x'_i = (x_i)^2 : i \in [m] \rangle$ (m squares) (191)

Private input of P :

$\langle x_i : i \in [m] \rangle$ (m square-roots) (192)

Commit. For $j \in [s']$

$P : w_j \leftarrow^{\$} \mathbb{G}$ (mask) (193)

$P \rightarrow V : w'_j = (w_j)^2$ (194)

Challenge. $P \leftrightarrow V$:

$c \leftarrow^{\$} (\{0,1\}^m)^{s'}$ (coin-toss s' m-bit vectors) (195)

Respond. For $j \in [s']$:

$P \rightarrow V : r_j = w_j * (*_{i \in [m]} (x_i)^{c_{j,i}})$ (196)

Verify. For $j \in [s']$:

V : If $(r_j)^2 \neq w'_j * (*_{i \in [m]} (x'_i)^{c_{j,i}})$, then REJECT (197)

V : ACCEPT (198)

Fig. 13: **ZKPoK of vector of square-roots.**

ZKPoK of vector of square-roots consistent with vector of classes (see Fig. 14).

- **Setup.** As common input, the parties know a vector of bits (199) and a vector of squares (200). As private input, P knows a respective vector of square roots whose classes are consistent with the vector of bits (201).
- **Procedure.** For each challenge index, P selects as mask a random encoding of bit 0 (202) and commits to it by sending its square to V (203). As each challenge, the parties decide a random subset of the squares (encoded as a vector of bits, with the positions with value 1 representing the selected squares) (204). As response, P sends to V a square root of the masked version of the product of the respective subset of squares (205). V verifies that the square-root is correct (206) and that it encodes the XOR of the bits whose positions were selected for verification (207). V accepts the proof if all responses are correctly verified (208).
- **Remarks.** Compared with the previous ZKPoK of square-roots, the essential difference now is that P does not intend to hide the classes of the known square-roots. As will be shown ahead, in §F.2, this ZKPoK may be useful even if P knows the trapdoor and thus can use it to extract all the square-roots known by V.

Common input of P and V :

$$\langle d_i \in \{0,1\} : i \in [m] \rangle \ (m \text{ classes}) \tag{199}$$

$$\left\langle x'_i = (x_i^{(d_i)})^2 : i \in [m] \right\rangle \ (m \text{ squares}) \tag{200}$$

Private input of P :

$$\left\langle x_i^{(d_i)} : h(x_i) = d_i, i \in [m] \right\rangle \tag{201}$$

(m square-roots with consistent classes)

Commit. For $j \in [s']$:

$$P : w_j \xleftarrow{\$} h^{-1}(0) \ (\text{mask class } 0) \tag{202}$$

$$P \to V : w'_j = (w_j)^2 \tag{203}$$

Challenge. $P \leftrightarrow V$:

$$c \xleftarrow{\$} (\{0,1\}^m)^{s'} \ (\text{coin-toss } s' \ m\text{-bit vectors}) \tag{204}$$

Respond. For $j \in [s']$:

$$P \to V : r_j = w_j * (*_{i \in [m]} (x_i)^{c_{j,i}}) \tag{205}$$

Verify. For $j \in [s']$:

$$V : \text{If } (r_j)^2 \neq w'_j * (*_{i \in [m]} (x'_i)^{c_{j,i}}), \text{ then } \textsc{Reject} \tag{206}$$

$$V : \text{If } h(r_j) = \oplus_{i \in [m]} (c_i \cdot d_i) \tag{207}$$

$$V : \textsc{Accept} \tag{208}$$

Fig. 14: **ZKPoK of vector of square-roots consistent with vector of classes.**

ZKPoK of equivalent decommitments of UB and UH BitComs (see Fig. 15).

- **Setup.** Both parties know the public parameters of a XOR-homomorphic UH-BitCom scheme (209) and of a XOR-homomorphic UB-BitCom scheme (210). As common input, the parties also know a vector of UH-BitComs (211) and a vector of UB-BitComs (212), both with the same length. As private input, P knows a respective vector of bits (213) and respective vectors of decommitments (encodings) of the UH BitComs (214) and UB BitComs (215). The group-operations do not mix elements from different schemes, because it is assumed that the groups underlying each type of BitCom scheme are different.
- **Procedure.** For each challenge index, P selects a random bit-mask (216) and then uses the two BitCom schemes to commit to the bit mask, namely with an UH-BitCom (217) (218) and an UB-BitCom (219) (220). As each challenge, the parties decide a random subset of the positions inside the vector (the subset is encoded as a vector of bits, with the positions with value 1 representing the selected positions) (221). As response, P sends to V the masked XOR of the bits in the selected positions (222), and sends the masked product of the respective subset of decommitments for each BitCom

scheme (223) (224). V verifies that the decommitments are valid and consistent with the disclosed masked XOR of bits (225) (226). V accepts the proof if all responses are correctly verified (227).

– **Remark.** The protocol can be trivially extended to prove consistency across more than two BitCom schemes, independently of them being UB or UH (for example, this is useful when considering the optimization described in §E.2). For each challenge index: in the *commit* stage, P would commit the random bit-mask by sending one BitCom per respective BitCom scheme; in the *respond* stage, P would send the result of XORing the bit-mask and of all bits in the selected positions, and it would also send the decommitments, within the several respective BitCom schemes, of the homomorphic combination of BitCom in the challenged subset of positions; in the *verify* stage, V would verify that they all were valid decommitments of the same bit.

Common input of P and V.

Public parameters of BitCom schemes.

$$(\mathbb{G}_B, h_1) \text{ (for UH-BitComs)} \quad (209)$$

$$(\mathbb{G}_A) \text{ (for UB-BitComs)} \quad (210)$$

$$\langle x'_i = (x_i)^2_B : i \in [m] \rangle \text{ (UH BitComs)} \quad (211)$$

$$\left\langle y'_i = (-1)^{b_i}_A (y_i)^2_A : i \in [m] \right\rangle \text{ (UB BitComs)} \quad (212)$$

Private input of P.

$$\langle b_i \in \{0,1\} : i \in [m] \rangle \text{ (bits)} \quad (213)$$

$$\langle x_i \in h^{-1}(b_i) : i \in [m] \rangle \text{ (square-roots)} \quad (214)$$

$$\langle y_i \in \mathbb{G}_A : i \in [m] \rangle \text{ ((pseudo-)square-roots)} \quad (215)$$

Commit. For $j \in [s']$:

$$P : d_j \xleftarrow{\$} \{0,1\} \text{ (bit-mask)} \quad (216)$$

$$P : w_j \xleftarrow{\$} h_B^{-1}(d_j) \text{ (encoding for UH BitCom)} \quad (217)$$

$$P \rightarrow V : w'_j = (w_j)^2_B \text{ (UH BitCom of } d_j) \quad (218)$$

$$P : z_j \xleftarrow{\$} \mathbb{G}_A^{-1}(0) \text{ (encoding for UB BitCom)} \quad (219)$$

$$P \rightarrow V : z'_j = (-1)^{d_i}_A (z_j)^2_A \text{ (UB BitCom of } d_j) \quad (220)$$

Challenge.

$$c \xleftarrow{\$} (\{0,1\}^m)^{s'} \text{ (coin-toss } s' \text{ } m\text{-bit vectors)} \quad (221)$$

Respond. For $j \in [s']$:

$$P \rightarrow V : e_j = d_j \oplus_{i \in [s']} b_i \cdot c_{j,i} \quad (222)$$

$$P \rightarrow V : r_j = w_j *_B (*_{B\, i \in [s']} (x_i)^{c_{j,i}}) \quad (223)$$

$$P \rightarrow V : t_j = z_j *_A (*_{A\, i \in [s']} (y_i)^{c_{j,i}}) \quad (224)$$

Verify. For $j \in [s']$:

$$V : (r_j)^2_B =^? w'_j * (*_{i \in S_j} x'_i) \wedge e_j =^? h_B(r_j) \quad (225)$$

$$V : (-1)^{e_j}_A (t_j)^2_B =^? z'_j * (*_{i \in S_j} y'_i) \quad (226)$$

V : If all verifications OK, then Accept, else Reject (227)

Fig. 15: **ZKPoK of equivalent decommitments of different BitComs schemes.** The protocol is exemplified for one UH-BitCom scheme and one UB-BitCom scheme, but the protocol could be trivially adjusted to any combination of BitCom schemes (with XOR-homomorphic properties), just adapting the formalism to the respective sampling, committing and decommitment procedures. It is also trivial to adjust the protocol to proving equivalence between more than 2 BitCom schemes.

ZKP of n-out-of-m square-roots encoding bit 0 (see Fig. 16).

– **Setup.** As common input, the parties know a vector of UH-BitComs (squares) (228). As private input, P knows a subset of vector positions (229) for which it knows decommitments of bit 0 (i.e., square-roots in class 0) (230). The goal of the ZKPoK is for P to knowledge of decommitments to 0 in at least a certain number of positions, but without revealing anything about the subset of positions or anything else about the number of positions for which it knows a decommitment of 0. It is worth noticing that this setting does not prevent P from knowing decommitments of more UH-BitComs, for bits 0 or 1.

– **Procedure.** For each challenge index, P produces a permuted tuple of masked BitComs, as follows: it selects a random permutation of the positions of the vector (231); then permutes the vector of squares accordingly (232); then selects masks for all positions, as random encodings of bit 0 (233); then squares the masks (234); then

Common input:

$$P, V : X' = \left\langle x'_j \equiv (x_j^{(0)})^2 : j \in [m] \right\rangle \tag{228}$$

Private input of P:

$$P : J \subseteq [m] : \#(J) = n \tag{229}$$

$$P : X_J = \left\langle x_j^{(0)} : j \in J \right\rangle \tag{230}$$

(satisfying $h(x_j^{(0)}) = 0 : j \in J$)

Commit. For $k \in [s']$:

$$P : \mathit{\Pi}_k \xleftarrow{\$} \mathrm{Perm}(\langle 1, ..., m \rangle) \tag{231}$$

$$P : y'_{k,j} = x'_{\mathit{\Pi}_k(j)} : j \in [m] \tag{232}$$

$$P : z_{k,j}^{(0)} \xleftarrow{\$} h^{-1}(0) : j \in [m] \tag{233}$$

$$P : z'_{k,j} = (z_{k,j}^{(0)})^2 : j \in [m] \tag{234}$$

$$P \to V : w'_{k,j} = z'_{k,j} * y'_{k,j} : j \in [m] \tag{235}$$

Challenge.

$$P \leftrightarrow V : c \xleftarrow{\$} \{0,1\}^{s'} \quad \text{(coin-toss } s' \text{ bits)} \tag{236}$$

Respond. For $k \in [s']$:

If $c_k =^? 0$ (*reveal permutation and masks*) :

$$P \to V : \mathit{\Pi}_k \tag{237}$$

$$P \to V : z_{k,j}^{(0)} : j \in [m] \tag{238}$$

If $c_k =^? 1$ (*reveal square-roots of* n *committed squares*) :

$$P \to V : L_k = \mathit{\Pi}_k(J) \tag{239}$$

$$P \to V : r_{k,l} = z_{k,l}^{(0)} * x_{L_k(l)} : l \in [n] \tag{240}$$

Verify. For $k \in [s']$:

If $c_k =^? 0$ (*verify permutation and masks*) :

$$V : \mathit{\Pi}_k \in^? \mathrm{Perm}(\langle 1, ..., m \rangle) \tag{241}$$

$$V : h(z_{k,j}) =^? 0 : j \in [m] \tag{242}$$

$$V : (z_{k,j}^{(0)})^2 * x'_{\mathit{\Pi}_k(j)} =^? w'_{k,j} : j \in [m] \tag{243}$$

If $c_k =^? 1$ (*verify square-roots*) :

$$V : L_k \subseteq^? [m] \wedge \#(L_k) =^? n \tag{244}$$

$$V : h(r_{k,j}) =^? 0 : j \in [n] \tag{245}$$

$$V : (r_{k,l})^2 =^? w'_{L_k(l)} : l \in [n] \tag{246}$$

V : If all verifications OK, then ACCEPT, else REJECT (247)

Fig. 16: **ZKP of n-out-of-m square-roots class 0.** The protocol can be easily adapted with a *random seed checking* technique to reduce the number of communicated group-elements to just those responded for bit-challenges with value 1, i.e., to an expected number $n \cdot s'/2$.

multiplies the vector of squared masks with the permuted vector of original squares and sends the result to V (235). As each challenge, the parties decide a random bit (236). If the challenge is 0, then P shows that the permutation was well constructed, by revealing the permutation (237) and all the masks encoding bit 0 (238). If the challenge is 1, P reveals a permuted subset (of size consistent with the goal of the proof) of positions for which it knows decommitment of bit 0 (239), and reveals the respective square-roots encoding bit 0 (240). V verifies the responses given by P. If challenge is 0, V verifies that the committed vector was well constructed, by verifying: that the revealed permutation is valid (241), that the revealed group elements are encodings of bit 0 (242), and that they are indeed the masks that lead the permuted vector of squares to the committed vector of squares (243). If challenge is 1, V verifies that P knows the correct amount of square-roots encoding bit 0, by verifying: that the revealed subset with of positions is valid (244), that the revealed group-elements encode bit 0 (245), and that they are indeed square-roots of the committed squares in the respective positions (246). V accepts the proof if all responses are correctly verified (247).

– **Remarks.**
  - **RSC.** A type of *random seed checking* (RSC) technique can be implemented inside this protocol. The committed permuted masked squares can be generated based on small random seeds and the common input. In the *commit* stage, P only needs to send short RSC commitments of the generated elements. Then, the responses associated with challenge 0 are simply replaced by the respective seeds, whose communication cost can be neglected. Finally, for each challenge with value 1, the number of group elements is equal to the vector length (square-roots for the positions of the selected subset, squares for the remaining positions).
  - **Proving bit-wise relations.** An example of application of the described ZKPoK is proving that three committed bits satisfy a bit-wise AND relation. This is possi-

ble because such relation can be represented as the existence of 2-out-of-3 bits with value 0 in a certain triplet that can be generated, from the original 3 BitComs, using only the available XOR-homomorphism (see (248)). The application can be extended to prove a NAND relation, by homomorphically XOR'ing the bit 1 (i.e., negating) to all the initial BitComs (249). Furthermore, the ZKPoK can also be trivially adapted to prove conditions expressible as *at least a certain number of bits with a certain value*. For that purpose, the logic of permutation and masking is maintained, changing only the bit values that P reveals in the *respond* stage. This allows a direct proof of the condition mentioned in §6.2, for proving a NAND relation, formally expressed in (250).

$$b_3 = \text{AND}(b_1, b_2) \Leftrightarrow 0 \in^2 \langle b_1 \oplus b_3, b_2 \oplus b_3, b_3 \rangle \tag{248}$$
$$b_3 = \text{NAND}(b_1, b_2) \Leftrightarrow 0 \in^2 \langle b_1 \oplus b_3 \oplus 1, b_2 \oplus b_3 \oplus 1, b_3 \oplus 1 \rangle \tag{249}$$
$$b_3 = \text{NAND}(b_1, b_2) \Leftrightarrow 1 \in^2 \langle b_1 \oplus b_3, b_2 \oplus b_3, b_3 \rangle \tag{250}$$

F.2 Prove correct Blum Integer

Before making use of the UH and UB BitCom schemes required in the S2PC-with-BitComs protocol, both parties need to be assured that they are correctly parametrized. The following paragraphs look into the particular case of proving that a certain number is a Blum integer, as this type of integers can be used to instantiate the BitCom schemes. First, some basic properties of Blum integers are reviewed; then, some problems of overlooking a proof of their correctness are identified; finally, two efficient ways of achieving such a proof are described (the second is new).

Properties of Blum integers. A Blum integer is defined as a positive integer composed only of the product of powers of two primes, where each prime is congruent with 3 modulo 4 and each power has an odd exponent. A more specialized definition could be just the integers composed as the simple product of two primes congruent with 3 modulo 4, but, as shown ahead, for the security of the S2PC-with-BitComs it is enough to consider the general definition. Properties of Blum integers can be found in many textbooks of introduction to number theory (e.g., [NZM91]). The respective multiplicative group is defined as a group set endowed with a group operation, with the former being the set of non-negative integers which are co-prime to and lower than the Blum integer, and the later being multiplication modulo the Blum integer (henceforth simply denoted *multiplication*). Group elements (also denoted as *residues*) referring to them as residues, any group element out-of-bounds is to be considered modulo the Blum integer.

Blum integers have interesting properties that distinguish them from other integers. In particular, they are the only moduli that simultaneously satisfy the following 2 properties:

- *Each quadratic residue has exactly 4 distinct square-roots, two of which for each possible Jacobi Symbol (-1 and 1).* Since the Jacobi Symbol is multiplicative, it is easier to consider the *class* of each square-root as the respective homomorphism to XOR, i.e., class 0 means Jacobi Symbol 1, and class 1 means Jacobi Symbol -1.
- *The residue -1 is non-quadratic and has Jacobi Symbol 1 (i.e., class 0).* This means that the additive inverse of a group element is a different group element with the same square and the same class. The two elements trivially correlated in this manner have different least significant bit (i.e., one is even and the other is odd) when represented as

a non-negative integer lower than the modulus. Thus, for example, it may be assumed that the *proper* element is the one whose least significant bit is equal to its class (when dealing with square-roots it is important to define a *proper* square root, out of the two possible with the same square and same class). It is trivial to adjust group multiplication to only operate on *proper* elements.

Two other useful properties related with finding square-roots are noticed: (i) the knowledge of two square-roots of different class enables an efficient discovery of the prime factorization of the Blum integer (upon a simple computation of a modular sum and the computation of a greatest common divisor, e.g., using the Euclidean algorithm); (ii) the knowledge of the prime factorization enables efficient discovery of all 4 square roots of any quadratic residue. If a party knowing the factorization of a Blum integer is able to assume that it is infeasible, for the other party, to find the factorization and to decide quadratic residuosity of random residues, then the modulus can be used as an instantiation basis for several components of the S2PC-with-BitComs protocol.

– **UB-BitCom scheme.** A square of a random residue is an UH BitCom of the class of the residue, because each square has square-roots in both classes; computational *binding* derives from the (assumed) inability of the comitter party to factor the Blum integer (knowledge of two square-roots of different classes would enable factorization).
– **UB-BitCom scheme.** A random residue in class 0 is an UB BitCom of its quadratic residuosity character (0 if quadratic residue, 1 if non-quadratic residue). To commit to a 0 or a 1 the committer selects a random residue and then respectively sends its square or the additive inverse of the square (both of which have class 0). The *hiding* property derives from the assumed inability of the receiver (who must not know the factorization of the Blum integer) to decide quadratic residuosity.
– **2-out-of-1 OT.** Since each square has one and only one proper square root of each class, the knowledge of the trapdoor enables extraction of exactly 2 elements, out of which the party that generated the square (and who does not know the factorization of the Blum integer) only knows one (the one that it used to generate the square).
– **Forge-and-lose.** Assuming intractability of deciding quadratic residuosity, the UH and the UB BitCom schemes can share the same trapdoor. This means that P_A can encrypt (i.e., UB commit) her input bits with a scheme with the same trapdoor as the scheme with which the bits of P_B are UH committed.

Note: The knowledge of the factorization of a Blum integer is computationally equivalent to the knowledge of a non-trivial square-root of 1 (i.e., a square-root of 1 in class 1) in the respective multiplicative group. Any residue multiplied by this value is converted into a new residue with different class but same square.

Problems with non Blum integers. The S2PC-with-BitComs becomes insecure if the suggested instantiation based on Blum integers is actually sustained on a non Blum integer. The need to prove correctness of the proposed Blum integer is now motivated, showing some security problems that may arise if such proof is not given.

A Blum integer is congruent with 1 modulo 4 (and this ensures that -1 has class 0). Since this can be verified non-interactively, without knowing the factorization, the attention can be restricted to integers of this form. These are all odd integers that have an even number of (including none) prime factors congruent with 3 modulo 4 exponentiated to an odd exponent. In other words, -1 remains class 0 regardless of the number of prime factors that are congruent with 1 modulo 4, and of the prime powers that have an even exponent. The use of non-Blum integers of this form would raise security problems.

For example, things could go wrong with moduli with exactly 3 prime factors, with at least one of them being congruent with 3 modulo 4 and such that this prime's higher power (dividing the Blum integer) would have odd exponent. Toy examples would be integers 105 and 1755. In the respective multiplicative groups, each quadratic residue has exactly 8 square-roots, 4 of which in each possible *class*. Since the additive inverse of a square-root is also a square-root in the same class, the 4 square-roots in a certain class can be grouped into 2 pairs, each pair being a trivially correlated pair of square-roots. Any 2 square-roots not in the same pair are non-trivially correlated, i.e., cannot be found except by being able to find some non-trivial factor of the modulus. This would raise at least two problems in the S2PC-withBitComs protocol:

– The 2-out-of-1 OT is not possible (though a 4-out-of-1 OT is), since for each square there are now 4 possible proper square-roots, 2 for each class.
– For correctly constructed connectors of input wires of $\mathrm{P_B}$, $\mathrm{P_A}$ would, in the Verify stage, respond correctly to *verification* challenges. However, in the later Evaluate stage, a malicious $\mathrm{P_A}$ could respond to *evaluation* challenges by disclosing multipliers with correct homomorphic properties that would nonetheless lead the known input BitComs into group-elements different from those that were used to derive the circuit input keys. The problem is that the squaring operation is no longer a permutation from the domain of proper square-roots in a certain class onto the set of squares, but it is rather a non-injective function.

Another problematic case is that of a moduli that once divided by the highest square factor does not retain any prime factor that is congruent with 3 modulo 4. Toy examples are integers 325 and 585. In the respective multiplicative groups, each quadratic residue either has all square-roots with class 0 or all square-roots with class 1. This means that the respective 2-to-1 square BitCom scheme would become UB, such that the receiver of the BitComs would be able, with the trapdoor, to decrypt the underlying bits – this would be a privacy problem.

Proving correctness of Blum integers. If the factorization of a modulus is known, it is trivial to verify that it is a Blum integer. The challenge is in how to prove correctness without giving away the factorization (or any other non-trivial information). Two methods are described below, whereby a prover (P), knowing the factorization of a modulus, convinces a verifier (V), not knowing the factorization, that the proposed modulus is a Blum integer. The protocols constitute ZKPoKs of the factorization of the Blum integer. (The informal titles below are not standard, but just intend to be suggestive of the respective method.).

– **The *ask for (pseudo-)square-roots in random classes* method.** An efficient method is from van de Graaf and Peralta [vdGP88], based on the observation that a modulus is a Blum integer if and only if it is not a square, not the power of a prime, is congruent with 1 modulo 4, and every residue in class 0 has a square-root or a pseudo square-root[25] of each class. The protocol starts with V verifying that the proposed modulus is congruent with 1 modulo 4, is not a square and is not the power of a prime. Then, the parties perform a two-party coin-tossing to decide (or

[25] A *pseudo square-root* of an integer is intended to mean a square-root of the additive inverse of the integer.

receive from a trusted source of randomness) a vector with random residues in class 0 (with square-root unknown by V), and a respective random vector of bits. Finally, P computes and reveals a vector of square-roots or pseudo-square-roots with classes consistent with the respective vector of random bits. V accepts the claim if and only if all received elements are square roots in the expected classes (261). The number of bits of statistical security (soundness) is equal to the vector length.

– **The *ask for known square-roots* method.** This paragraph introduces a new method (see Fig. 17), making use of the ZKPoKs previously defined, and avoiding the coin-tossing of group-elements (though still requiring the coin-tossings used to decide challenges in the ZKPoKs). For a given Blum integer modulus (251) for which P knows the trapdoor (252), V starts by verifying that the modulus is congruent with 1 modulo 4 (253). In a first phase, P gives a ZKPoK of non-trivial square-root of 1, as described in Fig. 12 (this is done by showing ability to take square-roots of different classes) (254). In a second phase, V sends a vector of random bits (i.e., classes) to P (255), then selects a respective vector of random encodings (random group-elements in the respective class) (256) and then sends the respective squares to P (257). V gives a ZKPoK of square-roots in those classes (258), as described in Fig. 14 (notice the exchanged roles). Finally, P computes the square-roots with respective classes and sends them to V(259). If any square-root is different from the one (or its additive inverse) known by V, then V rejects the claim that the modulus is a Blum integer (260); otherwise V accepts the claim (261).

Common input of P and V :

$(\mathbb{G} \equiv \mathbb{Z}_N^*, \times(\text{mod}N), h)$ (group specification) (251)

$(N = p_1^{a_1} p_2^{a_2} : p_1 \neq p_2 \wedge p_1, p_2 \in \text{Primes})$

$(a_1 \equiv a_2 \equiv 1(\text{mod } 2), p_1 \equiv p_2 \equiv 3(\text{mod } 4))$

$(\forall x : h(x) = (1 - \text{JS}_N(x))/2)$

Private input of P :

$t = \text{NTSQRT1}_N$ (i.e., $(h(t) = 1 \wedge t^2 = 1)$ (252)

Procedure.

V : If $n \neq 1(\text{mod } 4)$, then REJECT (253)

$P(t) \leftrightarrow V$: $\text{ZKPoK}_{\text{NTSQRT1}}(N)$ (see Fig. 12) (254)

$V \rightarrow P : b_i \overset{\$}{\leftarrow} \{0,1\} : i \in [s']$ (random bits) (255)

$V : \mu_i^{(b_i)} \overset{\$}{\leftarrow} h^{-1}(b_i) : i \in [s']$ (random encodings) (256)

$V \rightarrow P : \mu'_i = (\mu_i^{(b_i)})^2(\text{mod } N) : i \in [s']$ (257)

$V(\mu_{[s']}) \leftrightarrow P : \text{ZKPoK}_{\text{Sqrts-with-classes}}(b_{[s']}, \mu'_{[s']})$ (258)

(see Fig. 14, but notice the exchanged roles)

$P \rightarrow V : u_i^{(b_i)} = \text{SQRT}[t]^{(b_i)}(\mu'_i) : i \in [s']$ (259)

V : For $i \in [s']$:

  If $u_i^{(b_i)} \neq \pm\mu_i^{(b_i)} \vee h(u_i^{(b_i)}) \neq b_i$ then REJECT (260)

V : ACCEPT (261)

Fig. 17: **ZKPoK of Blum integer trapdoor ($\text{ZKPoK}_{\text{BI-trap}}$).**

Explanation of the* ask for known square-roots *method. If phase 1 is accepted by V, then with overwhelming probability every quadratic residue has square-roots in both classes; consequently, there is at least one prime factor congruent with 3 modulo 4 whose higher power that divides the modulus has an odd exponent. In particular, this excludes the possibility of the modulus being a square (because there are square-roots with Jacobi-symbol -1). Also, this excludes the possibility of the modulus being a prime: it cannot be a prime modulus congruent with 1 modulo 4, because in that case each square would only have square-roots in the same class; it cannot be a prime congruent with 3 modulo 4 because it was verified that the modulus is congruent with 1 modulo

4. Then, if P sends, in phase 2, all proper square-roots that V already knew, then with overwhelming probability the modulus has exactly only two prime factors.[26]

Since V discloses the class of the known square-roots and gives a respective ZKPoK, a honest P is ensured that by revealing those square-roots it does not give away any new information to a possibly malicious V*. Since, after a successful phase 1, V knows that each quadratic residue has square-roots in both classes, the ZKPoK of known vector of square-roots is indeed ZK. Specifically, even if the modulus is not a Blum integer, in case P* is malicious, multiplication by a random square-root class 0 is a permutation within the group of elements class 0. This means that the square-root revealed after each challenge is uniformly distributed over all residues of the known class, and thus does not reveal information about which proper square-root (out of possibly several proper square-roots) V knows.

Remark. From the point of view of correctness, it does not matter which odd exponents are applied to each of the two prime factors. Still, from the point of view of hiding the trapdoor, it is to the best interest of the party selecting the modulus that the Blum integer is given by the simple product of two large primes with roughly the same size (i.e., same number of bits). This is to make more difficult the factorization of the integer by known methods, whose complexity increases with the size of the smallest factor. Thus, a proof for general Blum integers is sufficient and adequate, even though there are known proofs (more costly) to show that a certain integer is a Blum integer composed simply as the product of two primes [TLL03].

Complexity. Both methods require P to compute square-roots, each of which can be obtained after one exponentiation modulo each prime factor, and then applying the Chinese remainder theorem. In terms of communication the methods differ.

The *ask for known square-roots* method as described requires, per round (i.e., per component of the vector of challenges), the communication of 6 group elements, corresponding to 2 for the ZKPoK of trapdoor, 1 square selected by V, 2 for the ZKPoK of square-roots with consistent classes, and 1 square-root disclosed by P. By applying a RSC technique to the ZKPoK of trapdoor, the overall (expected) number of communicated group-elements can be reduced to only 4.5 per round.

The *ask for (pseudo-)square-roots in random classes* method as described requires performing two-party coin-tossing to obtain enough random bits to extract, for each round, one residue class 0, and sending, for each round, one group element (the square-root or pseudo square-root). Even though a random element is in class 0 with probability 50%, any random element with class 1 can be converted to a random element with class 0, after a simple multiplication by a fixed residue of class 1 (e.g., defined as the shortest residue in class 1). If the coin tossing is done without random seeds, then each party is required to exchange the equivalent to (an expected value of) one group element per round, which overall in the protocol sums up to 3 group elements communicated per round. However, for the simulability of this coin-tossing sub-protocol, only the simulated P needs to be able to enforce the outcome. Thus, even though P has to send her contribution without compression, V can simply send a commitment to a short seed and then open the commitment to reveal the seed (e.g., see Fig. 7 but with reversed roles). Thus, overall the number of communicated group-elements can be lowered to 2 per round.

[26] Interestingly, this method has some *forge-and-lose* flavor – if P tries to cheat with a modulus with more than two prime factors, then V learns at least part of the factorization.

G Proof of Security

This section proves the security of the 1-output S2PC-with-BitComs protocol, within the ideal/real simulation paradigm [Can00] (assuming rewinding capability). §G.1 defines the ideal functionality of 1-output S2PC with commitments, intermediated by a *trusted third party* (TTP), where only one party ($\mathrm{P_B}$) learns a circuit output, and both parties obtain random commitments of the input and output of both parties, and with each party learning the decommitment to her own respective input and output. §G.2 describes a simulator for the case of each possible malicious party, assuming it has rewindable access to the malicious party in the real world and has the capability to delay message delivery and abort executions in the ideal world. When the simulator plays the role of one malicious party in the ideal world, it induces a joint probability distribution of outputs in the ideal world that is indistinguishable from the respective distribution in the real world. §G.3 has a proof of soundness of the output of a honest $\mathrm{P_B}$ against a malicious $\mathrm{P_A^*}$. This soundness property is used as argument in the description of the simulator. Across this section, p is an index that can be replaced by A or B, and the following notation is used: $\widehat{\mathrm{P}}_p$ denotes the honest party in the ideal world, $\widehat{\mathrm{P}}_p^*$ denotes the malicious party in the ideal world, P_p denotes the honest party in the real world, P_p^* denotes the malicious party in the real world, $\mathrm{P}_{\bar{p}}$ denotes the party that is not P_p.

G.1 Ideal functionality

In the *ideal* functionality, a TTP intermediates the communication between the two parties (see Fig. 18). As part of the setup, it is assumed that the two original parties and the TTP have agreed on the Boolean circuit that needs to be evaluated and on the parameters of the commitment schemes (262). Each of the two parties has a private input (263) and starts the ideal protocol by sending it to the TTP (264). The ordering between the two messages is irrelevant, i.e., it does not matter which party sends the input in first place. Then, the TTP computes the circuit output (265) and the commitments and respective decommitments of the circuit inputs and circuit output ((266),(267),(268)). Finally, the TTP sends the expected output first to $\mathrm{P_A}$ (269) and then to $\mathrm{P_B}$ (270). In particular, $\mathrm{P_A}$ learns the commitments and decommitments of her own circuit input, and the commitments of the circuit input and circuit output of $\mathrm{P_B}$. Respectively, $\mathrm{P_B}$ learns the commitments and decommitments of his own circuit input and circuit output, and the commitments of the circuit input of $\mathrm{P_A}$.

If at any point, during the protocol execution, the TTP receives an ABORT message from one party (271), then it relays it to the other party and exits the execution (272). If a party, before finishing its execution in the protocol, receives an ABORT message from the TTP (273), then it exits outputting FAIL (274). Thus, in this idealized version, a malicious $\widehat{\mathrm{P}}_\mathrm{A}^*$ is able to first receive the output from the TTP and then decide whether or not to prevent the ideal $\widehat{\mathrm{P}}_\mathrm{B}$ from receiving the respective output from the TTP. This ability is not symmetric, in the sense that a malicious $\widehat{\mathrm{P}}_\mathrm{B}^*$ is not able to do the same.

Remark. A different idealized version could be defined, in order that a malicious ideal $\widehat{\mathrm{P}}_\mathrm{B}^*$ could also learn his result and only then decide whether or not to prevent a honest ideal $\widehat{\mathrm{P}}_\mathrm{A}$ from obtaining the respective result. For example, this could be achieved with the TTP sending the outputs concurrently (instead of first to $\mathrm{P_A}$ and only then to $\mathrm{P_B}$), and allowing the malicious ideal $\widehat{\mathrm{P}}_\mathrm{B}^*$ to delay the message delivery to $\widehat{\mathrm{P}}_\mathrm{A}$, until deciding

Setup. Common input:

$\widehat{\mathrm{P}}_{\mathrm{A}}, \widehat{\mathrm{P}}_{\mathrm{B}}, \mathrm{TTP} : C, \mathscr{C}^{B}_{UH}, \mathscr{C}^{A}_{UH}$ (262)

Private input of $\widehat{\mathrm{P}}_p$

$\mathrm{P}_p : x_p, \text{ for } p \in \{A, B\}$ (263)

Send inputs to TTP.

$\widehat{\mathrm{P}}_p \rightarrow TTP : x_p, \text{ for } p \in \{A, B\}$ (264)

TTP local computation.

$\mathrm{TTP} : y_B = C(x_A, x_B)$ (265)

$\mathrm{TTP} : \langle \underline{x}_A, \bar{x}_A \rangle \leftarrow^{\$} \mathscr{C}^{B}_{UH}[x_A]$ (266)

$\mathrm{TTP} : \langle \underline{x}_B, \bar{x}_B \rangle \leftarrow^{\$} \mathscr{C}^{A}_{UH}[x_B]$ (267)

$\mathrm{TTP} : \left\langle \underline{y}_B, \bar{y}_B \right\rangle \leftarrow^{\$} \mathscr{C}^{A}_{UH}[y_B]$ (268)

Send output to parties.

$TTP \rightarrow \widehat{\mathrm{P}}_{\mathrm{A}} : \langle \underline{x}_A, \bar{x}_A \rangle, \bar{x}_B, \bar{y}_B$ (269)

$TTP \rightarrow \widehat{\mathrm{P}}_{\mathrm{B}} : \bar{x}_A, \langle \underline{x}_B, \bar{x}_B \rangle, \left\langle y_B, \underline{y}_B, \bar{y}_B \right\rangle$ (270)

Concurrent process.

TTP : If receiving $\perp$ from P_p, (271)

then send $\perp$ to $\mathrm{P}_{\bar{p}}$ and exit (272)

TTP : If receiving $\perp$ from TTP, (273)

then output Fail and exit (274)

Fig. 18: **Ideal functionality of 1-output S2PC with committed inputs and outputs. Legend:** $\widehat{\mathrm{P}}_p$ (party in the ideal world, with $p \in \{A, B\}$); C (Boolean circuit specification, implicitly defining the domain of circuit-inputs and circuit-outputs of both parties); $\mathscr{C}^{p}_{UH}$ (unconditionally hiding commitment scheme used to commit the private input and output of $\mathrm{P}_{\bar{p}}$ – the scheme is labeled with the index p, in order to match the notation used in the S2PC-with-BitComs protocol, where the trapdoor is known by P_p); $\leftarrow^{\$} \cdot$ (random sampling from domain $\cdot$); $\bar{\cdot}$ (commitment of $\cdot$); $\underline{\cdot}$ (decommitment, i.e., encoding necessary to reveal $\cdot$).

whether or not to drop it. This version would be less restrictive regarding the Decide UH-BitCom Permutations stage in the real world, so it is more interesting to show that the protocol can achieve the more difficult version where P_{A} receives her output first. In particular, it needs to be ensured that P_{A} is the first to know the result of the BitCom permutations, in the Decide UH-BitCom Permutations stage. Nonetheless, in the case of some 2-output extensions of the protocol, with both parties receiving a circuit output, it is natural to change the ideal functionality, either for $\widehat{\mathrm{P}}_{\mathrm{A}}$ to receive first the output from the TTP (e.g., for 2-output via single-path execution, where only P_{B} evaluates GCs), of for both parties to receive the output concurrently (e.g., for 2-output via dual-path execution).

G.2 Simulators

Henceforth, S^*_p is used to denote a simulator with black-box rewinding capable access to a real and possibly malicious P^*_p, and with access to play a single instance of an ideal protocol while impersonating an ideal malicious $\widehat{\mathrm{P}}_p$, with pause-play-stop capability over the TTP. S^*_p will be constructed such that the joint probability of views of the output of both parties in the ideal world is indistinguishable from the respective joint probability distribution in the real world, with P^*_p being malicious, and with $\mathrm{P}_{\bar{p}}$, $\widehat{\mathrm{P}}_{\bar{p}}$ and TTP being honest. The relation between the two worlds is illustrated in Fig. 19, for the case of $\mathrm{S}^*_{\mathrm{A}}$.

Preliminary remarks.

- In typical C&C-GCs based protocols without BitComs, it is common, in the proof of security against a malicious $\mathrm{P}^*_{\mathrm{A}}$, that the simulator $\mathrm{S}^*_{\mathrm{A}}$ extracts the input bits of $\mathrm{P}^*_{\mathrm{A}}$ from the rewinding and replaying of several C&C phases. Usually, $\mathrm{S}^*_{\mathrm{A}}$ discovers the underlying bit corresponding to the input keys received in the C&C *verification* stage, and then uses the rewinding capability to receive some of those keys again but for an *evaluation* stage, thus learning what are the bits of $\mathrm{P}^*_{\mathrm{A}}$. In contrast, in the proof that follows the simulator extracts the input bits directly from the ZKPoKs associated with decommitments of BitComs.

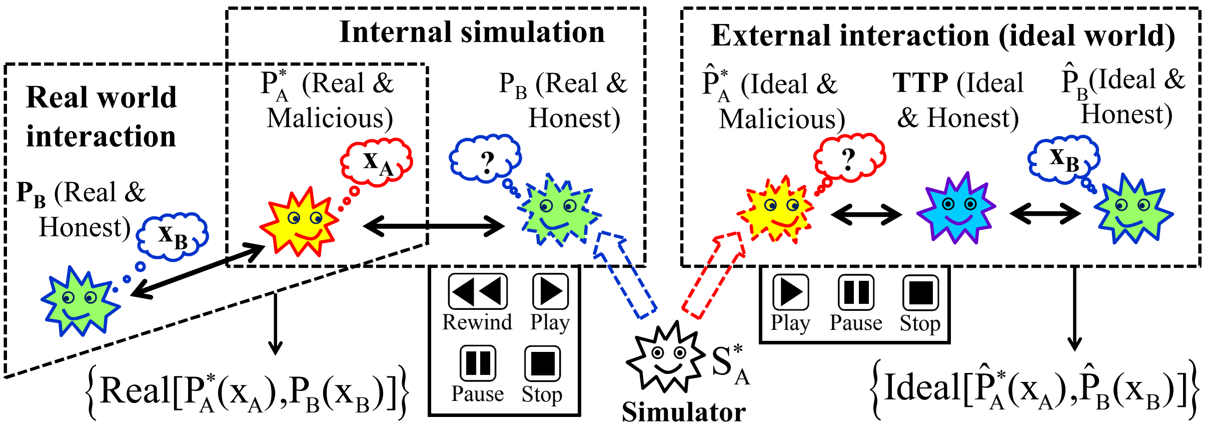

Internal simulation
External interaction (ideal world)
Real world interaction
P_B (Real & Honest)
x_B
P_A^* (Real & Malicious)
x_A
P_B (Real & Honest)
?
$\hat{P}_A^*$ (Ideal & Malicious)
?
TTP (Ideal & Honest)
$\hat{P}_B$(Ideal & Honest)
x_B
Rewind
Play
Pause
Stop
Play
Pause
Stop
S_A^*
Simulator
$\{Real[P_A^*(x_A),P_B(x_B)]\}$
$\{Ideal[\hat{P}_A^*(x_A),\hat{P}_B(x_B)]\}$

Fig. 19: **Ideal/real simulation.** Example of simulator S_A^*, used to prove security against a malicious P_A^*. The simulator S_A^* controls an ideal (malicious) $\widehat{P}_A^*$ interacting with a TTP in an ideal interaction, and having *pause*, *play* and *stop* capability over the TTP, but without being able to *rewind* it (the suggestive terms mean that S_A^* can delay message delivery and it can stop the execution by sending an Abort message to the TTP). The input x_B of the ideal $\widehat{P}_B$ is unknown to S_A^*. As an auxiliary computation, S_A^* makes an internal simulation of a honest P_B interacting with a real malicious P_A^*. S_A^* has rewindable black-box access to P_A^*; i.e., S_A^* can *rewind*, *pause*, *play* and *abort* interactions with P_A^* (the suggestive terms mean that S_A^* can fully control the time flow of the simulation, forward and backward, including behaving differently when *playing* after some *rewinding* action). In the beginning of the internal simulation, S_A^* does not know the private input x_A of P_A^*, but it may learn it during the simulation, namely by exercising its rewinding power. The goal of the proof of security is to show that the joint distributions of output are indistinguishable between the real and ideal worlds.

– In the proof of security, the remarks about a party aborting an execution are meant to encompass both the acts of proactively aborting an execution (e.g., halting an interaction) or of taking some other action that leads the other (honest) party to abort. For example, it is implicitly considered that sending malformed messages (that would not be accepted by the other party and would thus lead this other party to abort, outputting Fail) are considered a form of abort.

G.2.1 Simulator S_A^* for a malicious P_A^*

1. **Extract trapdoor and circuit input of P_A^*.** S_A^* simulates a honest P_B from the beginning of the protocol, with a random input (of adequate length), interacting in the real world with a malicious P_A^*. As the simulation proceeds, if P_A^* aborts on the first time it reaches a new step of the protocol, then S_A^* sends Abort to the TTP (thus leading the ideal $\widehat{P}_B$ to output Fail), and the ideal $\widehat{P}_A^*$ outputs whatever P_A^* outputs. Otherwise, if P_A^* does not abort in the first simulated execution until a certain step, there is a non-negligible probability that it will also not abort during subsequent attempts upon rewinding and retrying the simulation until such step. Thus, S_A^* uses the rewinding capability to extract the trapdoor of P_A^* from the ZKPoK of trapdoor (45), and extract the input bits (and respective encodings) of P_A^* from the ZKPoK of equivalent decommitments of UB and UH BitComs (54).[27] S_A^* finishes this process in an iteration for which P_A^* does not abort the execution until the end of the Produce initial BitComs stage.

[27] If P_A^* has shown some willingness to participate in the protocol, by not aborting in the first attempt, then the rewinding process can be done in expected polynomial time. Using a technique of estimating the probability of abort, from [GK96], it is possible to ensure that the simulator runs in expected polynomial time, even in cases where the probability of abort is neither negligible nor noticeable. For simplicity, this detail is ignored henceforth, assuming that the probability estimation and respective modifications to the simulator are implicit in the rewinding process whenever necessary.

2. **Begin coin-tossing of random group-element permutations.** S^*_A resumes the interaction in the real world, through the COMMIT and CHALLENGE stages and entering the DECIDE UH-BITCOM PERMUTATIONS stage (Fig. 9), passing through the step where P^*_A commits to her contribution (117) and where P^*_A gives a respective ZKAoK of known decommitment (118), but pausing immediately before P_B would send his contribution to P^*_A (120). If P^*_A aborts on the first attempted execution, e.g., failing to give a proper ZKAoK of known decommitment, then S^*_A sends ABORT to the TTP (thus leading the ideal $\widehat{\mathrm{P}}_\mathrm{B}$ to output FAIL), and the ideal $\widehat{\mathrm{P}}^*_\mathrm{A}$ outputs in the ideal world whatever P^*_A outputs in the real world. Otherwise, if the execution has not aborted, S^*_A uses the rewinding capability to learn the contribution of P^*_A to the coin-tossing sub-protocol.[28]
3. **Interact with the TTP.** S^*_A, impersonating the ideal $\widehat{\mathrm{P}}_\mathrm{A}$, sends to the TTP the private circuit input x_A of P^*_A. The TTP, which meanwhile also accepts the input from the ideal $\widehat{\mathrm{P}}_\mathrm{B}$, computes locally the circuit output $y_B = C(x_A, x_B)$ and the BitComs of all inputs and outputs, and the respective decommitments. Then, the TTP sends to S^*_A (impersonating the ideal $\widehat{\mathrm{P}}^*_\mathrm{A}$) the BitComs of the input and output bits of $\widehat{\mathrm{P}}_\mathrm{B}$, and the BitComs and decommitments of the input bits of $\widehat{\mathrm{P}}_\mathrm{A}$. S^*_A pauses the interaction of the ideal world, before the TTP sends the respective output to $\widehat{\mathrm{P}}_\mathrm{B}$.
4. **Enforce the BitCom values selected by the TTP.**
   (a) **Determine necessary outcome of the coin-tossing sub-protocol.**
      - **For each input wire of $\mathbf{P_A}$.** S^*_A has received from the TTP the final UH BitCom and the respective final encoding corresponding to the input bit of P^*_A. Furthermore, S^*_A had already extracted the initial encoding known by P^*_A. Thus, S^*_A multiplies the final encoding with the inverse of the initial encoding, thus determining the group element (the random permutation, encoding of 0) that the coin-tossing sub-protocol needs to output (for this wire).
      - **For each input wire of $\mathbf{P_B}$.** S^*_A has received from the TTP only the final UH BitCom (square), but not the respective decommitment (proper square-root). However, since S^*_A has already extracted the trapdoor of P^*_A, it can compute the two respective proper square-roots (one of which is the final encoding that the ideal $\widehat{\mathrm{P}}_\mathrm{B}$ will receive in case the ideal execution ends successfully). S^*_A had also already selected one initial encoding (as part of playing the role of P_B with a random input (47)). S^*_A selects as final encoding the proper square-root (of the square received by the TTP) that has the same class as the selected initial encoding. Then, S^*_A determines the multiplier (encoding of 0) that leads the initial encoding into the final encoding (by multiplying the final encoding with the multiplicative inverse of the initial encoding) – this is the permutation that the coin-tossing sub-protocol must output (for this wire).
      - **For each output wire of $\mathbf{P_B}$.** S^*_A has received from the TTP only the final UH BitCom (square), but not the respective decommitment (square-root). S^*_A multiplies the final UH BitCom with the multiplicative inverse of the initial UH BitCom (previously received from P^*_A (56)), thus obtaining the square of what must be the permutation decided by the coin-tossing sub-protocol. Using the trapdoor of P^*_A, S^*_A extracts the respective proper square-root encoding bit

[28] In the exemplified coin-tossing protocol (Fig. 9) (and respective instantiation based on El-Gamal encryption): S^*_A extracts the decommitments after several rewindings of the respective ZKAoK (118) (respectively, obtains the El-Gamal encryption key from a ZKPoK of discrete-log and uses it to decrypt the contributions of P^*_A from the respective El-Gamal encryptions that are serving as UB commitments (117)).

0 – this is the permutation that the coin-tossing sub-protocol must output (for this wire) .

Since the encodings selected by the TTP are uniformly distributed across the group elements of a certain class, in the respective groups, the permutation that leads each initial encoding into the respective final encoding is also uniformly distributed (within the group elements of class 0), regardless of the initial encodings being uniformly random or not.

(b) **Determine the contribution by $\mathrm{P_B}$.** Since $\mathrm{S}^*_{\mathrm{A}}$ already knows the contribution that $\mathrm{P}^*_{\mathrm{A}}$ selected in the initial part of the coin-tossing sub-protocol, $\mathrm{S}^*_{\mathrm{A}}$ determines what needs to be the contribution of $\mathrm{P_B}$, in order for the outcome of the coin-tossing sub-protocol to be the permutations (across all input and output wires) calculated in the previous step. Since the intended permutations are uniformly distributed, from within the set of all possible permutations, so is the contribution determined by $\mathrm{S}^*_{\mathrm{A}}$ (impersonating $\mathrm{P_B}$), regardless of the distribution used by $\mathrm{P}^*_{\mathrm{A}}$ to select her contribution. Assuming, as described in §D, that each contribution is a vector of group elements of class 0 (one group element per wire, in the respective group), and the combination of both contributions corresponds to multiplying both group elements, for each wire, then $\mathrm{S}^*_{\mathrm{A}}$ computes the contribution by $\mathrm{P_B}$, for each wire, as the product between the final permutation and the inverse of the contribution of $\mathrm{P}^*_{\mathrm{A}}$.

(c) **Continue coin-tossing of random group-element permutations.** At this point, $\mathrm{P}^*_{\mathrm{A}}$ in the real world is in the equivalent step of the ideal world in which $\widehat{\mathrm{P}}^*_{\mathrm{A}}$ has already received her output from the TTP, but $\widehat{\mathrm{P}}_{\mathrm{B}}$ has not yet received his output. $\mathrm{S}^*_{\mathrm{A}}$ resumes the DECIDE UH-BITCOM PERMUTATIONS stage, in the role of $\mathrm{P_B}$, sending to $\mathrm{P}^*_{\mathrm{A}}$ the calculated complementary contribution of $\mathrm{P_B}$ (119).

5. **Finalize the execution.** $\mathrm{S}^*_{\mathrm{A}}$ continues the interaction of the DECIDE UH-BITCOM PERMUTATIONS stage and into the RESPOND stage, receiving from $\mathrm{P}^*_{\mathrm{A}}$ all information that a real $\mathrm{P_B}$ would receive. If $\mathrm{P}^*_{\mathrm{A}}$ aborts or if any verification related with the VERIFY stage fails, then $\mathrm{P_B}$ in the real world would output FAIL. In this case, $\mathrm{S}^*_{\mathrm{A}}$ sends ABORT to the TTP (leading $\widehat{\mathrm{P}}_{\mathrm{B}}$ to also output FAIL), and $\widehat{\mathrm{P}}^*_{\mathrm{A}}$ in the ideal world outputs whatever $\mathrm{P}^*_{\mathrm{A}}$ outputs in the real world. Otherwise, if the VERIFY stage is successful, then $\mathrm{S}^*_{\mathrm{A}}$ resumes the ideal protocol execution, letting the TTP send the output awaited by $\widehat{\mathrm{P}}_{\mathrm{B}}$. Finally, $\widehat{\mathrm{P}}^*_{\mathrm{A}}$ in the ideal world outputs whatever $\mathrm{P}^*_{\mathrm{A}}$ outputs in the real world.

   Remark. With overwhelming probability, the joint probability distribution of the outputs of the ideal parties ($\mathrm{S}^*_{\mathrm{A}}$ and $\widehat{\mathrm{P}}_{\mathrm{B}}$) is indistinguishable from the joint probability distribution of the real parties, whenever the real honest $\mathrm{P_B}$ has the same circuit input as the ideal $\widehat{\mathrm{P}}_{\mathrm{B}}$. A rare exception would be if the real $\mathrm{P_B}$ could not compute a correct final output, after arriving to the EVALUATE stage. However, this could only happen with probability negligible in the number of challenges (in case $\mathrm{P}^*_{\mathrm{A}}$ acted maliciously), as shown in the modular proof of soundness in §G.3 (and complemented by the probability calculation in §A). In particular, it is shown that if $\mathrm{P_B}$ accepts the VERIFY stage, then there is a negligible probability (denoted soundness error probability) that $\mathrm{P_B}$ fails to compute the correct output.

G.2.2 Simulator $\mathrm{S}^*_{\mathrm{B}}$ for a malicious $\mathrm{P}^*_{\mathrm{B}}$

1. **Extract trapdoor and circuit input of P^*_B.** *Ipsis verbis* to the description of S^*_A, but interchanging the indices A and B, and extracting the circuit input from the respective ZKPoK of decommitments (step 49, detailed in Fig. 13). At this point, the real execution is paused at the end of the PRODUCE INITIAL BITCOMS stage.
2. **Begin coin-tossing of random group-element permutations.** S^*_B resumes the interaction in the real world, through the COMMIT and CHALLENGE stages and entering the DECIDE UH-BITCOM PERMUTATIONS stage, until the end of the step where P^*_B sends his contribution to P_A (119), and immediately before P_A reveals the random permutations resulting from combination of the contributions of the two parties (120) (this would reveal to P^*_B what was the contribution of P_A). If P^*_B aborts on the first attempted execution, then S^*_B sends ABORT to the TTP (thus leading the ideal $\widehat{\mathrm{P}}_\mathrm{A}$ to output FAIL), and the ideal $\widehat{\mathrm{P}}^*_\mathrm{B}$ outputs in the ideal world whatever P^*_B outputs in the real world. Otherwise, if the execution has not aborted, then S^*_B has learned the contribution of P^*_B, but P^*_B still does not know the contribution of P_A.[29]
3. **Interact with the TTP.** S^*_B in the role of the ideal $\widehat{\mathrm{P}}^*_\mathrm{B}$ sends to the TTP the private circuit input of P^*_B (previously extracted). The TTP then computes and sends the outputs to the respective parties. $\widehat{\mathrm{P}}_\mathrm{A}$ receives her output in first place, consisting only of BitComs of all input and output bits, and decommitments of the input bits of $\widehat{\mathrm{P}}_\mathrm{A}$; i.e., it does not contain any circuit output bits. Then, S^*_B (in the role of the ideal $\widehat{\mathrm{P}}^*_\mathrm{B}$) receives from the TTP the BitComs of the circuit input bits of $\widehat{\mathrm{P}}_\mathrm{A}$, the BitComs and decommitments of the circuit input bits of $\widehat{\mathrm{P}}^*_\mathrm{B}$ (associated with the input bits extracted and used by the real P^*_B), and the circuit output bits of $\widehat{\mathrm{P}}^*_\mathrm{B}$ and respective BitComs and decommitments.
4. **Enforce the BitCom values selected by the TTP.**
   (a) **Determine necessary outcome of the coin-tossing sub-protocol.** S^*_B computes what needs to be the outcome permutations of the coin-tossing sub-protocol, so that the initial BitComs (and respective decommitments) are transformed into the final BitComs (and respective decommitments) decided by the TTP in the ideal world. This is accomplished similarly to how S^*_A did it in the respective part of the simulation, but making the necessary adjustments. Specifically for the input wires of P_A, S^*_B has received from the TTP only the UH BitComs, but not the respective encodings, so it needs to make use of the trapdoor of P^*_B to compute proper square-roots. For the input and output wires of P_B the procedure is easier, involving only computing multiplications and multiplicative inverses, because .
   (b) **Determine the contribution by P_A.** S^*_B computes what needs to be the contribution of P_A in the real world, such that the combination with the contribution of P^*_B results in the intended permutations. This is accomplished in the same way as S^*_A did it in the respective part of the simulation.
   (c) **Continue coin-tossing of random group-element permutations.** S^*_B then resumes the DECIDE UH-BITCOM PERMUTATIONS stage, taking advantage of the

[29] This is a crucial point of the simulation. On one hand, S^*_B already has the capability to influence the result of the coin tossing, because it knows the contribution of P_B and because it can use the rewinding capability to make P_B accept any (false) final result of the coin tossing. On the other hand, S^*_B does not yet know what values the TTP will choose. In this situation, it is specially delicate the fact that the TTP sends the output first to $\widehat{\mathrm{P}}_\mathrm{A}$ and only then to $\widehat{\mathrm{P}}_\mathrm{B}$. In particular, once S^*_B (impersonating the ideal $\widehat{\mathrm{P}}^*_\mathrm{B}$) receives the output from the TTP, the simulation must guarantee that P_A in the real world receives the respective output, even if P^*_B misbehaves. However, since P_A still has to interact with P^*_B, it becomes defined that P_A outputs the final BitComs even if P^*_B decides to abort the execution after this point (e.g., by ignoring the messages sent by P_A).

simulability of the coin-tossing to convince P_B^* to accept a (fake) final permutation equal to the one selected by the TTP.[30]

If P_B^* aborts at some point in this process, then the real P_A would not abort. Instead, a real P_A would output the final encodings of his input bits, and the final UH BitComs of the input and output bits of both parties, resulting from applying the permutations obtained from the coin-tossing protocol (120) (and respective squares) to the initial encodings and BitComs, respectively ((109), (110)). This is consistent with the ideal $\widehat{\mathrm{P}}_\mathrm{A}^*$ having already received her output from the TTP. Conversely, if P_B^* behaved in a valid way, then at this point a real P_A would simply send more messages to P_B and finish the interaction with a valid output, with no more room for P_B^* to affect the outcome of P_A. However, S_B^* still has to rewind.

5. **Produce new GCs.** S_B^* rewinds the execution until the moment, in the COMMIT stage of the protocol, where the GCs and connectors are built. For the indices previously selected for *evaluation* (in the CHALLENGE stage), S_B^* builds new GCs with the same topological position of garbled gates and wires, but with the circuit computing the constant function that outputs the circuit output $\widehat{\mathrm{P}}_\mathrm{B}^*$ received from the TTP.[31] For the remaining indices, S_B^* reuses the same GCs and connectors. Then, using the rewinding capability, S_B^* forces the coin-tossing that decides the C&C vector of challenges to lead to the same outcome Since the new *evaluation* GCs (and respective) connectors are indistinguishable from the correct GCs, the simulator can use the rewinding capability repeatedly until the protocol reaches again the end of the DECIDE UH-BITCOM PERMUTATIONS stage, after having enforced the same C&C partition as in the first execution, but now having built incorrect elements for the evaluation challenges.
6. **Finalize the execution.** Subsequently, S_B^* (still impersonating P_A) sends to P_B^* all the responses of the RESPOND stage, namely with the *connectors* of wires of P_A being related with the simulated random input (of adequate length) of P_A. Finally, S_B^* lets $\widehat{\mathrm{P}}_\mathrm{B}^*$ in the ideal world output whatever P_B^* outputs in the real world. In particular, if P_B^* decides to do a correct evaluation of GCs and connectors, it will obtain a circuit output consistent with the what the TTP has determined in the ideal world.

G.2.3 Additional remarks

On the hiding property of garbled gates. In the description of the simulator (S_B^*) for the case of a malicious P_B^*, it was assumed that garbled gates hide the underlying Boolean gate. This allowed S_B^* (in the role of P_A) to construct fake GCs that computed a constant function, without P_B^* being able to distinguish them from correct GCs – this is a common proof technique for GC based protocols (e.g., see [LP09]). It is now shown why and how the hiding assumption of the garbled gates may be relaxed.

One conceivable example of garbling scheme that does not satisfy the mentioned assumption is one that produces GCs with verifiable correctness. This verifiable property

[30] In the exemplified coin-tossing protocol (Fig. 9), S_B^* would send to P_B^* the vector of final group elements (120), and then, in the next step, would use the rewinding capability to make P_B^* accept the false ZKA (123) that the revealed outcome is consistent with the contributions committed by P_A and contributions sent by P_B^*. In the suggested instantiation based on El-Gamal encryption, this would correspond to use the rewinding capability to maliciously succeed in ZKPs of Diffie-Helman tuples.

[31] This is a common proof technique (e.g., see [LP09]), assuming the GC hides the underlying Boolean circuit. As explained in §G.2.3, it is possible to prove security under a weaker assumption, namely that the learned keys hide the underlying bits, even though each garbled gate may reveal the underlying Boolean operation.

would not, on its own, resolve all security problems (e.g., selective failure attack related with the input bits of $\mathrm{P_B}$), but it could be seen as a security upgrade. For example, it would prevent a malicious $\mathrm{P_A^*}$ from making $\mathrm{P_B}$ accept the output of the evaluation of a Boolean circuit different from what was agreed, despite potential problems still related with $\mathrm{P_A^*}$ giving incorrect input keys to $\mathrm{P_B}$. Another conceivable example is a GC that discloses some information about the underlying Boolean gates, but does not guarantee their correctness. More specifically, it could be that each garbled gate could be verified for the fact that either: it is a correct garbling of the intended Boolean gate; or it simply cannot be degarbled (e.g., it would output an Error symbol if evaluated with an incorrect ley). In this later example, the selective failure attack could be applied also at the intermediate wire level, but $\mathrm{P_A^*}$ would still be prevented from making $\mathrm{P_B}$ learn the result of evaluating a different Boolean circuit.

This possibly-desirable verifiability property of GCs (i.e., that the garbled gates have the correct underlying Boolean operation) would prevent making the assumption (in the proof of security) of indistinguishability with GCs that compute a constant function. This is not a problem to the S2PC-with-BitComs protocol defined in this paper, because it is possible to relax the assumption of the hiding property of garbled gates, to assuming that the output keys of the garbled gates (also including the circuit output keys) do not reveal the respective underlying bits. It is worth emphasizing that now it must be assumed that the circuit output keys do not directly reveal their output keys (in evaluation GCs), whereas previously it could be exceptionally allowed that the circuit output keys could reveal the underlying bit. In the new proof of security, the simulator $\mathrm{S_B^*}$ simply embeds the necessary *fakeness* in the output *connectors*, leaving all the GCs unchanged and correct. For each output wire, the fake widgets lead both output keys into the same output bit encoding (inner encoding). In particular, each widget can be a short ciphertext, resulting from encrypting a short seed (using the respective circuit output key as encryption key). By having the two widgets of each wire be a pair of two different ciphertexts encrypting the same seed, it would be guaranteed that the two different circuit output keys would lead into the same inner encoding (a group element resulting from a pseudo-random expansion of the seed).

Simulators for the optimized 1-outut protocol. Simulability also holds in the described optimized protocol. In the optimization allowing *short UH BitComs of P_A* (§E.2), security derives from the properties of the added ZKPoKs and (in terms of simulability) from the fact that the new elements related with short BitComs do not belong to the output of a honest party. In the RSC technique (§E.1), the only significant changes of information flow in the protocol occur in the Commit and Respond stages, but without interfering with the logic of the explained simulators. Thus, it is possible to have a proof of security of the optimized protocol somewhat similar to the one described for the non-optimized protocol. Still, it is interesting to notice that a certain instantiation of the RSC commitments allows a proof of security somewhat changed, namely with a simulator $\mathrm{S_B^*}$ that does not need to rewind in the Commit stage to rebuild GCs when interacting with a malicious $\mathrm{P_B^*}$. First, for simplicity it is assumed that the several RSC commitments sent in the Commit stage are replaced by a single short RSC UH commitment, based on the UH-BitCom scheme with trapdoor known by $\mathrm{P_B}$. For example, the commitment functionality could first apply a collision resistant hash, and then each of the output bits of the hash would be committed by an UH BitCom. In this case, in the Commit stage the simulator $\mathrm{S_B^*}$ would just send random squares (one for each bit of the hash output), before even building the GCs and connectors. This vector of squares would be the RSC commitment. Then, once the parties reach the RESPOND stage, S_B^* (in the role of P_A) simply sends (to P_B^*) random RSC seeds for verification challenges, and fake GCs and associated connectors for evaluation challenges. Then, S_B^* expands the RSC seeds into verification GCs and associated connectors, joins in the elements associated with evaluation challenges, computes the hash of all these elements Finally, S_B^* uses the trapdoor (previously extracted from P_B^*) to decommit this hash output from the respective RSC commitment. Interestingly, in this case S_B^* does not even have to be able to anticipate the outcome of the C&C challenge vector, which means the respective coin-tossing could also be simpler.

Simulators for 2-output extensions. This paper has not given a full description of protocol extensions that achieve 2-circuit-output functionalities, but has hinted on how to achieved them, via single-path and via dual-path execution approaches. There are corresponding adjustments needed to accomplish the respective proofs of security.

– **Non-fair 2-output via single-path.** The ideal functionality is changed so that the TTP sends the output first to P_B (the GC evaluator) and only then to P_A. Correspondingly, the role of the parties in the DECIDE UH-BITCOM PERMUTATIONS stage is exchanged, and the stage is partitioned in two parts, such that P_A only learns the permutations after P_B has learned his own S2PC protocol output (i.e., circuit output, encodings and UH BitComs).
– **Non-fair 2-output via dual-path.** The ideal functionality is changed so that the TTP sends the output concurrently to both parties.
– **Fair 2-output.** The ideal functionality is extended to consider the gradual release stage (e.g., see the resource fairness of the commit-prove-fair-open functionality proposed in [GMPY06]). There are several options to consider in terms of ideal functionality, namely whether the fair delivery is related only with circuit outputs, or also with their commitments and decommitments. Interestingly, the instantiation based on Blum integers considered for the S2PC-with-BitComs protocol is naturally suited to known protocols of gradual release. Thus, an extension for fair 2-output can be set to correspond essentially to adding the gradual release of a single group element, for each party.

G.3 Soundness against P_A^*

Soundness against a malicious P_A^* is only a partial aspect of security, but its isolated proof helps focusing on specific aspects of the statistical nature of the C&C challenges. To complete the argument given in the last step of the simulator S_A^*, it needs to be proven that a malicious P_A^* cannot make a honest P_B accept an incorrect final circuit output, unless perhaps with a negligible probability in the security parameters. If P_B detects malicious behavior from P_A and aborts safely without jeopardizing its own security, then the FAIL output of P_B is considered correct. For example, P_B cannot prevent a malicious P_A from aborting the execution of the protocol, not even in the ideal functionality, so the FAIL is not accounted as incorrect. Given that BitComs are part of the output, soundness also requires that P_B only accepts the BitComs of P_A if P_A indeed knows the respective decommitments. This is verified directly from the ZKPoK of equivalent decommitments between the UH and UB BitComs of the input bits of P_A (see Fig. 15), so henceforth it is simply assume that indeed P_A commits correctly to a single sequence of input bits.

Preliminary definitions

- *Incorrect element.* An element is said to be *incorrectly* generated only if it does not satisfy the prescribed relations with other elements. Conversely, an element is said to be *correct* even if its generation is deviated from the protocol specification by at most the usage of different probability distributions for sampling of related elements, but the sampling domains are correct and all deterministic transformations are preserved as prescribed. For example, a BitCom value is *correct* if the generator of the BitCom knows a valid decommitment, even if it was not selected uniformly at random.
- *Complainable inconsistency.* In a *selective failure* type-of-attack [MF06; KS06], a malicious P_A^* induces a certain error whose activation depends on the private input or output of P_B (or even of a certain bit value in an intermediate wire of the circuit). If special caution is not taken, then either: the detection by P_A^* of the predicate of activation vs. non-activation may lead to a breach of privacy of P_B; or the avoidance of complaining (to protect privacy) in case of unsuccessful evaluation may lead to a breach of soundness. An inconsistency detected by a honest P_B is said to be *complainable* if P_B is able to inform (i.e., *complain* to) P_A about the inconsistency without jeopardizing the security of the protocol. Even though *complaining* has not been made explicit in the 1-output protocol, *complainability* is still an important aspect to consider when envisioning applications that consider linked S2PC executions.
- *Forged elements.* An *incorrect but successfully verified* element is called a *forgery.* Henceforth, forgeries are distinguished in two types: (i) a response that would be accepted by P_B in the VERIFY stage, even though it would be incorrect from the point of view of the specification of the RESPOND stage – it is shown ahead that such responses cannot be built, except with negligible probability in the cryptographic security parameter; (ii) an element that would be detected as incorrect in the VERIFY stage, but is instead selected for *evaluation* and for that reason is not detected as incorrect during the EVALUATE stage – it is shown ahead that, even though these elements can be built, they cannot affect the output of P_B, except with negligible probability in the statistical security parameter.

Sketch. In the S2PC-with-BitComs protocol, all inconsistencies found up to the end of the VERIFY stage are *complainable*, even if referring to *evaluation* indices. Conversely, all inconsistencies found in the EVALUATE stage are *non-complainable*, and actually at this point there is no more interaction between the parties (except perhaps in the context of a broader protocol, e.g, with linked executions). The proof of soundness is accomplished in three steps:

1. First, it is shown in §G.3.1 that if the elements (GCs and *connectors*) sent in the COMMIT stage are incorrect and become associated with verification challenges, then they cannot withstand the VERIFY stage (i.e., they are detected as incorrect and P_B safely aborts), unless P_A breaks the cryptographic assumptions in the responses that it gives in the RESPOND stage.
2. Then, it is shown in §G.3.2 that P_A^* is not able to produce a *non-complainable bad* response respective to a *good* GC and connectors, except again if it could break the cryptographic assumptions. In other words, if for a certain challenge index P_A^* has acted honestly in the COMMIT stage, then P_A^* is not able to produce undetectably incorrect responses in the *revealing for evaluation* of the respective connector.

3. Finally, it is observed in §G.3.3 that $\mathrm{P_B}$ is able to determine a correct output if at least one GC and respective connectors lead $\mathrm{P_B}$ to obtain a correct output (i.e., decommitments of the respective output BitComs, encoding the correct output bits) Evaluate stage. Thus, soundness can be broken only if $\mathrm{P_A^*}$ produces incorrect elements (GCs and associated connectors) in all the indices that will be associated with verification challenges, and correct elements for all the remaining indices. §A shows that the probability of this event is negligible in the overall number of GCs, for several C&C partitioning methods.

G.3.1 Incorrect GCs and connectors. A possible strategy by a mailcious $\mathrm{P_A^*}$ is to somehow build incorrect GCs and connectors (i.e., built inconsistently with the specification of the Commit stage) capable of leading to inconsistencies in the final output bits of $\mathrm{P_B}$. To cause damage in the output of $\mathrm{P_B}$, the incorrect elements must necessarily be associated with *evaluation* indices, as only this type of challenge contributes to the computation of the final circuit-output-bits. Below, it is shown that such *incorrect elements* would always be detected if they were instead associated with a *verification* index.

– **GCs.** By assumption, the garbling scheme used to produce GCs (57) allows them to be fully verified, once two correct keys are known per circuit input wire (and possibly some additional revealed randomness) ((83) and (84)). Thus, any incorrect GC can be detected if selected for verification. Henceforth it is assumed that the GCs are correct.
– **Connectors for input wires of $\mathbf{P_A}$.** (Review Fig. 4.) In the Commit stage, the elements related with each input wire of $\mathrm{P_A}$ are: one *inner square* (committing a permuted input bit) (61); and a respectively permuted pair of commitments of the two input keys (64). In the Respond stage, $\mathrm{P_A}$ is supposed, for the *revealing for verification* mode, to show the pair of circuit input keys and the *multiplier* (which encodes the permutation bit) (73). First, given the assumed binding properties of the commitment scheme used to commit input wire keys, $\mathrm{P_A}$ cannot lie about the pair of input keys that have been committed. Upon these keys being revealed, (along with the others for input wires of $\mathrm{P_B}$), $\mathrm{P_B}$ can verify that the GC was correctly built, and determine the bits underlying each input key, thus knowing how the pair of commitments was permuted. Second, the square of the *multiplier* must lead the initial UH BitCom of input bit into the respective inner square (square of the encoding of the permuted bit) (78). By the properties of the UH-BitCom scheme, $\mathrm{P_A^*}$ could not known another multiplier (proper square-root) encoding the other bit, or otherwise it would have found the trapdoor of the BitCom scheme. $\mathrm{P_B}$ verifies that the bit encoded by the revealed *multiplier* is indeed the permutation bit (79) with which the pair of key-committed was permuted. In conclusion, if the connectors of input wires of $\mathrm{P_A}$ were not correctly generated (e.g., the committed keys were incorrect or permuted inconsistently with in respect to the committed permuted bit), they will not pass the verification associated with the *reveal for verification* mode.
– **Connectors for input wires of $\mathbf{P_B}$.** (Review Fig. 5.) In the Commit stage, the elements related with each input wire of $\mathrm{P_B}$ are two independent inner squares (69) that serve as one-way commitments of independent inner encodings of bits 0 and 1 (i.e., proper square-roots in classes 0 and 1), respectively. Possibly, there may also be additional elements (*widgets*) that convert the respective (still hidden) inner encodings into the respective (still hidden) circuit input keys (66). In the Respond stage, $\mathrm{P_A}$ is supposed, for the *revealing for verification* mode, to reveal an ordered pair of inner

encodings of (proper square-roots in class) 0 and 1, respectively for the first and second inner squares (74). Even though in this case $\mathrm{P_A}$ knows the trapdoor of the UH-BitCom scheme, the scheme is actually being used as an UB commitment scheme to the respective inner encodings (by definition each square only has one proper square-root in each class). Thus, $\mathrm{P_A}$ must reply correctly, or otherwise $\mathrm{P_B}$ will output FAIL in the VERIFY stage, when verifying the encoded bits and the square of the encodings (81). The circuit input keys are thus determined (eventually using widgets) by $\mathrm{P_B}$. With two keys per input wire, $\mathrm{P_B}$ can verify the correctness of the GC, verifying that the keys are indeed correct and have internal bits consistent with 0 and 1 ((83), (84)). Thus, to pass a verification associated with the *reveal for verification* mode, $\mathrm{P_A}$ must have committed correctly to the connectors related with input wires of $\mathrm{P_B}$.

– **Connectors for output wires of $\mathbf{P_B}$.** (Review Fig. 6.) In the COMMIT stage, the elements related with each output wire of $\mathrm{P_B}$ are two independent inner squares (69), serving as one-way commitments of independent inner encodings of bits 0 and 1, respectively. Possibly, there might also exist widgets to help convert circuit output keys (yet unknown by $\mathrm{P_B}$) into the respective inner encodings (also yet unknown by $\mathrm{P_B}$) (68). The verification of GCs using two keys per input wire leads $\mathrm{P_B}$ to obtain all possible output keys and learn their respective underlying bits (85). In the RESPOND stage, $\mathrm{P_A}$ is supposed, for the *revealing for verification* mode, (Possibly using widgets,) $\mathrm{P_A}$ is thus able to compute the candidate pairs of inner encodings (86), which should encode bits 0 and 1, in respective order, and whose square should be equal to the pairs of inner squares (87). Thus, to pass a verification associated with the *reveal for verification* mode, $\mathrm{P_A}$ must have committed correctly the connectors related with output wires of $\mathrm{P_B}$.

G.3.2 Correct GCs and connectors. In the COMMIT stage, $\mathrm{P_A}$ sends one GC and respective connectors for each C&C index. If an index is selected for *evaluation*, then $\mathrm{P_A}$ needs to send, in the RESPOND stage, the elements that correspond to the *revealing for evaluation* phase of the connectors. Even for *evaluation* challenges, $\mathrm{P_B}$ performs a verification associated with the *reveal for evaluation* mode of the connectors. It is now shown that if the GCs and connectors for a particular C&C index would have been validated by a *verification* challenge (i.e., if they were correct), then $\mathrm{P_A^*}$ is not capable of *forging* a response for an *evaluation* challenge. It is not being claimed that this would also be the case if the GC or respective connectors sent in the COMMIT stage were flawed from the beginning in a way that would not be validated by a *verification* challenge. The properties of forgeries are now examined:

– **Input wires of $\mathbf{P_A}$.** (Review Fig. 4.) For *evaluation* indices, $\mathrm{P_A}$ reveals the inner encoding of the permuted bit (75) and decommits the input key from the respective position of the permuted pair of commitments (76).
  - **Forged permuted bit.** Since it is assumed that the inner square is correct, the only possibility left for forgery related with the permuted bit would be for $\mathrm{P_A}$ to reveal an inner encoding of the complementary of the permuted bit. However, since it is being assumed that this GC and associated connectors would be validated if selected for a *verification* challenge, it must be the case that $\mathrm{P_A}$ would already be able to reply the inner encoding corresponding to the correct permuted bit. Thus, to forge the output, $\mathrm{P_A}$ would have to be able to break the binding property of the UH-BitCom scheme, finding a pair of non-trivially correlated square-roots. Henceforth it is assumed that $\mathrm{P_A}$ reveals the correct inner encoding.

- **Forged circuit input key.** Since it is assumed that the connector would pass the *verification* test, it follows that the key-commitments are correct. As described above, it is now assumed also that the revealed permuted bit is correct. Thus, the only possibility of forgery would be for $\mathrm{P_A}$ to reveal a different (incorrect) key that verifies well against the key-commitment (89). However, this would mean breaking the collision resistance (i.e., the binding property) of the commitment scheme.

Thus, if the GC and associated connectors would withstand a *verification* challenge, $\mathrm{P_B^*}$ is bound to respond correctly for the input wires of $\mathrm{P_A}$, or else be detected in a *complainable* condition.

– **Input wires of $\mathbf{P_B}$.** (Review Fig. 5.) In the COMMIT stage, $\mathrm{P_A}$ has sent two independent *inner squares* (in ordered position), for each input wire of $\mathrm{P_B}$ (69) (here assumed to be correct). In the RESPOND stage, for each possible bit value, $\mathrm{P_A}$ is supposed to reveal a *multiplier* (which encodes 0) (77) that leads the outer encoding of the bit (i.e., the decommitment of the input UH BitCom) into the respective independent *inner encoding* of the bit (a proper square-root of the respective *inner square*). For each possible bit value in each input wire, there is only one *multiplier* (proper square-root in the respective class), and its correctness can be homomorphically verified even without knowing any decommitment of the input BitCom. Thus, it is not possible for $\mathrm{P_A^*}$ to build forged multipliers for input wires of $\mathrm{P_B}$. Furthermore, it is being assumed that the elements sent in the COMMIT stage are all correct, the revealed multipliers allow $\mathrm{P_B}$ to obtain one correct input key per input wire of $\mathrm{P_B}$.

– **Output wires of $\mathbf{P_B}$.** (Review Fig. 6.) By assumption, the GC is correct and the obtained circuit input keys are correct. Thus, $\mathrm{P_B}$ is able to evaluate the GC to obtain one correct key per circuit output wire (96). Possibly with the help of (also assumed to be correct) *widgets*, $\mathrm{P_B}$ can find one inner encoding for only one of the two independent squares (97) (also assumed to be correct) that was sent by $\mathrm{P_A}$ in the COMMIT stage (69). Thus, the only possible forgery would be for $\mathrm{P_A^*}$ to reveal, in the RESPOND stage, incorrect multipliers. However, there is only one correct multiplier per bit value per output wire, and its correctness can be verified homomorphically, even without $\mathrm{P_B}$ knowing the respective inner encodings of the output wire, independent of the final circuit output bit of $\mathrm{P_B}$.

G.3.3 Decision of final output. Above, it was shown that incorrect commitments (GCs and associated connectors produced in the COMMIT stage) are detected as incorrect if they become associated with *verification* challenges, and that correct commitments either lead to a correct output (if the responses from the RESPOND are correct) or allow detection of incorrect responses. Thus, to lead $\mathrm{P_B}$ into accepting an incorrect output, $\mathrm{P_B}$ needs to produce incorrect commitments, and be lucky that none becomes associated with a verification challenge. Then, in order for the bad indices selected for evaluation to be validated by the *reveal for evaluation* mode, they need to lead $\mathrm{P_B}$ to find one valid decommitment of the UH BitCom of each output bit. Furthermore, by combining an undetected incorrect output with a correct output it is always possible to obtain the trapdoor of $\mathrm{P_A^*}$ (a pair of different decommitments to the same output BitCom; i.e., a pair of non-trivially correlated square-roots of the same square), and thus activate the forge-and-lose evaluation path to obtain a correct final output. Thus, the only way that $\mathrm{P_A}$ has to lead $\mathrm{P_B}$ to accept an incorrect output is to guess in advance exactly what indices will be selected for evaluation and then build incorrect elements for all these and only for these indices. As shown in §A, there is a negligible probability of such guess being correct, for practical C&C partition methods.

H Notation

Acronyms

- AES: *advanced encryption standard*
- BitCom: *bit commitment*
- C&C: *cut-and-choose*
- GC: *garbled circuit*
- JS: *Jacobi symbol*
- NAND: *(bit-wise) Negated AND operation*
- OT: *oblivious transfer*
- RSC: *random seed checking*
- S2PC: *secure two-party computation*
- SHA: *secure hash algorithm*
- TTP: *trusted third party*
- UB: *unconditionally binding*
- UH: *unconditionally hiding*
- XOR: *(bit-wise) eXclusive OR operation* (i.e., sum modulo 2)
- ZK: *zero-knowledge*
- ZKA: *ZK argument*
- ZKP: *ZK proof* (or generically meaning “ZKP or ZKA”)
- ZKAoK: *ZKA of knowledge*
- ZKPoK: *ZKP of knowledge*

Diverse symbols

- $\mathrm{P_A}$ (party A – the GC constructor)
- $\mathrm{P_B}$ (party B – the GC evaluator)
- p (index denoting the party: $p \in \{A, B\}$)
- x_p(input of P_p, in the ideal functionality)
- y_p (output of P_p, in the ideal functionality)
- b (bit value)
- c (bit index; e.g., underlying bit of key $k^{[c]}$, bit encoded by group element $\mu^{(c)}$, position of commitment $\bar{k}^{\langle c\rangle}$ in a pair, bit associated with a multiplier $\beta^{(0)}_{j,i,c}$ class 0 of an input or output wire of $\mathrm{P_B}$)
- $c_{j,i}$ (tentative output bit value in wire i of GC j, correct if $\mathrm{P_A}$ was honest)
- $\cdot \in^a \cdot$ (the value appearing at the left of the symbol occurs at least a times in the vector that appears on the right side of the symbol, e.g., $0 \in^2 \langle 0,1,0\rangle$ and $1 \in^2 \langle 1,1,0,1\rangle$)
- $\gtrapprox$ (greater than, or approximately equal)
- $\%$ (percent)
- $\lessapprox$ (less than, or approximately equal)
- $\oplus$ (XOR operation)

Symbols about C&C challenges

- s (total number of C&C challenges)
- e, v (number of *evaluation*, *verification* challenges)
- b, g (number of *bad*, *good* indices)
- j (index of challenge, often associated with a GC)
- J_E (subset of indices selected for *evaluation*)
- J_V (subset of indices selected for *verification*)
- J_{Ignore} (subset of indices ignored during the *evaluation* stage of the protocol, upon being detected as incorrect)
- $[s]$ (set $\{1, ..., s\}$)
- $\oslash$ (empty set)

Symbols about groups

- $\mathbb{G}$ (group-set)
- $*$ (group-operation, in multiplicative notation)
- $(\cdot)_p$ (identification that element inside parenthesis belongs to group $\mathbb{G}_p$, i.e., for which P_p knows the respective trapdoor)
- $(\cdot)^2_p$ (square of the element inside parenthesis, in group $\mathbb{G}_p$)
- $(\cdot)^{-1}_p$ (multiplicative inverse of the element inside parenthesis, in group $\mathbb{G}_p$)
- π (permutation bit ,used to permute input bits of $\mathrm{P_A}$)
- h (group homomorphism onto XOR)
- $h^{-1}(b)$ (subset of $\mathbb{G}$, containing the encodings of bit b used as decommitments in an UH-BitCom scheme)
- $\mathcal{U}'$ (UB BitCom of an input bit of $\mathrm{P_A}$)
- $\mathcal{U}$ (encoding used to generate $\mathcal{U}'$)
- t_p (trapdoor associated with $\mathbb{G}_p$, known by party P_p)
- $\#(\cdot)$ (size of set $\cdot$)
- $SQRT\,[t]^{(b)}_p\,(x)$ (using trapdoor t to compute a square-root class b of x, in group $\mathbb{G}_p$, for $p \in \{A, B\}$)
- $NTSQRT1_p$ (non-trivial square-root of 1, in $\mathbb{G}_p$)
- **Encodings in an UH-BitCom scheme.**
  - α (multiplier in $\mathbb{G}_B$, encoding of permutation bit π, used for input wires of $\mathrm{P_A}$)
  - β (multiplier in $\mathbb{G}_A$, encoding of bit 0, used for input and output wires of $\mathrm{P_B}$)
  - γ (encoding of 0, used to permute another encoding)
  - μ (initial encoding of input or output bit, before the final random permutation)
  - ν (inner encoding, used in connectors)
  - u, v (tentative encodings of μ and ν, respectively, correct if $\mathrm{P_A}$ was honest)
  - σ (final encoding of input or output bit, after the final random permutation)
- $\alpha', \beta', \gamma', \mu', \nu', \sigma'$ (squares of the respective encodings)

Symbols about circuits

- C (Boolean circuit specification)
- $GC_{\text{Build}}, GC_{\text{Eval}}, GC_{\text{Verify}}$ (algorithms for building, evaluating and verifying GCs)
- $\mathscr{C}$ (commitment used to commit wire-keys)
- $\mathcal{W}$ (widget that converts group elements into wire keys, or vice-versa)
- $\epsilon(\cdot)$ (function that reveals the bit underlying a circuit output wire key)
- i (index of wire)
- I_p (set of indices of input wires of P_p)
- $I_{A,B}$ (set of indices of input wires of $\mathrm{P_A}$ and $\mathrm{P_B}$)
- I_p (set of indices of input wires of P_p)
- InKeys (expression denoting a set of input keys)
- $\text{InKeys}_j^{(1)}$, $\text{InKeys}_j^{(2)}$ (congregations of 1 or 2, respectively, input key(s) per input wire of GC_j)
- IO_p (set of indices of input and output wires of P_p)
- IOkeys (same as InKeys, but referring to input and output keys)
- j (challenge index, often used as GC index)
- k (wire key – circuit input wire or circuit output wire)
- $\overline{k}$ (commitment of a wire key)
- $\underline{k}$ (randomness required to verify a decommitment of a wire key)
- $k^{\langle c\rangle}$ (key associated with position c in a pair of keys)
- $k^{[c]}$ (key with underlying bit c)
- O_p (set of indices of output wires of P_p)
- $\mathrm{O_{A,B}}$ (set of indices of output wires of $\mathrm{P_A}$ and $\mathrm{P_B}$)
- OutKeys (same as InKeys, but referring only to output keys)
- ξ (tentative value of a wire key, correct if $\mathrm{P_A}$ was honest)
- r_j (randomness possibly required to verify the correctness of a GC)

Symbols about protocols

- ALGS (algorithms and sub-protocols implicitly defined)
- $\mathrm{P}_p : \cdot$ (P_p makes computation $\cdot$)
- $\mathrm{P}_p \rightarrow \mathrm{P}_{\bar{p}} : \cdot$ (P_p sends message $\cdot$ to $\mathrm{P}_{\bar{p}}$)
- $\mathrm{P}_p \leftrightarrow \mathrm{P}_{\bar{p}} : \cdot$ (P_p and $\mathrm{P}_{\bar{p}}$ interact to obtain $\cdot$)
- $\mathrm{P}_p(a) \leftrightarrow \mathrm{P}_{\bar{p}}(b) : \mathrm{Prot}(c)$ (Protocol between P_p and $\mathrm{P}_{\bar{p}}$, where P_p has private input a, and $\mathrm{P}_{\bar{p}}$ has private input b, and c is a common input. For example, this notation is used for ZKPs, in Figs. 8, 9 and 10)
- $\leftarrow^{\$} \cdot$ (random sampling from domain $\cdot$)
- $\langle \underline{x}, \bar{x}\rangle \leftarrow^{\$} \mathscr{C}[x]$ (computation of a probabilistic commitment of x, returning a public part $\bar{x}$ – the commitment *per se* – and a private part $\underline{x}$ – the decommitment shown when revealing x)
- $\perp$ (Abort symbol)
- $\mathscr{C}_{\text{Verify}}(\mathscr{C}_y, x, \underline{x}, \bar{x})$ (verify correctness of decommitment $\langle x, \underline{x}\rangle$, against the commitment $\bar{x}$, when using commitment scheme $\mathscr{C}_y$.
- κ (cryptographic security parameter, also expressed as 1^κ)
- s' (statistical security parameter, also expressed as $1^{s'}$)
- $[s']$ (set $\{1, ..., s\}$, used as the set of challenge indices in ZKPs)

List of Figures

List of Tables