# Cross&Clean: Amortized Garbled Circuits With Constant Overhead

Jesper Buus Nielsen and Claudio Orlandi

Aarhus University, Denmark, {jbn,orlandi}@cs.au.dk

**Abstract** Garbled circuits (GC) are one of the main tools for secure two-party computation. One of the most promising techniques for efficiently achieving active-security in the context of GCs is the so called *cut-and-choose* approach, which in the last few years has received many refinements in terms of the number of garbled circuits which need to be constructed, exchanged and evaluated.
In this paper we ask a simple question, namely *how many garbled circuits are needed to achieve active security?* and we propose a novel protocol which achieves active security while using only a constant number of garbled circuits per evaluation in the amortized setting.

## 1 Introduction

Garbled circuits are one of the most widely used and promising tools for secure two-party computation. Garbled circuits were introduced by Yao [Yao82] and they were first implemented in Fairplay by Malkhi et al. [MNPS04].

The basic version of Yao's protocol only guarantees security in the presence of *passive* corruptions (i.e., when the adversary follows the protocol but might try to learn more information from its view). From a very high level point of view, since garbling schemes hide (to some extent) the circuit which is being garbled, a malicious party can garble a different function from the one he is supposed to without the honest party noticing it, therefore breaking the security of the protocol. During the years many approaches have been proposed to construct GC-based protocols with strong security guarantees against adversaries who deviate arbitrarily from the protocol (i.e., *malicious* or *active* corruptions). The main technique for achieving active security in the GC context is the so called *cut-and-choose* approach: in a nutshell, cut-and-choose involves several copies of the same circuit being garbled; afterwards, a random subset of the garbled circuits are checked for correctness, while the rest are evaluated.

There are many different instantiations of the cut-and-choose approach: in 2007, Lindell and Pinkas proposed a method which achieves security $2^{-\kappa}$ by garbling approximately $3\kappa$ copies of the circuit. This was improved in 2013 in several works [Lin13, HKE13, Bra13] using the so called *forge-and-lose* technique. In its most efficient instatiation [Lin13] this technique allows to achieve security $2^{-\kappa}$ using only $\kappa$ garbled circuits (by adding a "small" actively secure computation).

A different approach was taken by Nielsen and Orlandi in 2009 [NO09]. Using LEGO style cut-and-choose the overhead needed to achieve active security decreases logarithmically with the size of the circuit $f$ i.e., it is possible to get security $2^{-\kappa}$ using a replication factor of $O(\kappa/\log|f|)$. While the original LEGO approach required to perform exponentiations for each gate in the circuit, subsequent work [FJN+13, FJNT15] got rid of this limitation and only uses generic assumptions. A variant of the LEGO approach has proven itself particularly useful in the *amortized* setting i.e., when the two parties are evaluating the same circuit $f$ multiple times (say $\ell$) on different inputs. In this case the amortized overhead to get security $2^{-\kappa}$ is $O(\kappa/\log\ell)$ [LR14, HKK+14], and experimental validation achieves "blazing fast" results [LR15].

To summarize, while advanced styles of cut-and-choose techniques have shown that one can achieve practically-efficient actively-secure two-party computation in the amortized setting, in all of the above approaches the number of garbled circuits *grows linearly* with the security parameter. It is natural to ask whether this is an inherent limitation or whether it is possible to achieve actively-secure two-party computation based on garbled circuits *with constant overhead*.

### 1.1 Our Contribution

Before stating our contributions, it is important to clarify the question we are asking as much as possible: it is of course possible to achieve active security using a single garbled circuit and using the GMW compiler [GMW87] (i.e., proving in *zero-knowledge* that the circuit is well-formed). This is not a satisfactory solution since it is not *black-box* in the underlying garbling scheme and does not therefore preserve the efficiency of Yao's protocol. Jarecki and Shmatikov [JS07] proposed an instantiation of this paradigm using a specific number theoretic assumption (Pailler's cryptosystem [Pai99]): thanks to the algebraic nature of the underlying cryptosystem, in their approach the extra zero-knowledge proofs only add a constant overhead. We do not consider this solution satisfying either, since one of the strengths of Yao's protocol (both in terms of security and efficiency) is that it only requires to perform symmetric key operations per gate in the circuit. Therefore, we are only interested in solutions that can be instantiated using *any projective garbling scheme in a black-box way*. We are now ready to ask our question:

> *Can we achieve actively-secure two-party computation protocol in the amortized setting with only a constant overhead over Yao's protocol?*

We answer the question positively: let $p(\kappa)$ be an upper bound on the cost of generating, evaluating or checking a garbled gate[1], and let $A(\kappa)$ be a fixed function independent of the size of the circuit (this term describes the cost of

[1] In all known instantiations of garbled circuits generating a gate is the most expensive operation, requiring at most 4 calls to a PRF. Evaluating typically requires fewer calls while checking is equivalent to garbling.

performing some "small" fixed actively secure computation independent of the circuit). Then the amortized complexity of our protocol is bounded by:

$$O(1) \cdot |f| \cdot p(\kappa) + A(\kappa) \tag{1}$$

i.e., only a constant overhead over Yao's passive protocol and an additive factor independent of $|f|$.

### 1.2 Technical Overview

We give here a high level description of our techniques. We have two parties, Alice and Bob, respectively with inputs $\{x_i^A, x_i^B\}_{i\in[\ell]}$. At the end both parties should learn $y_i = f(x_i^A, x_i^B)$ for all $i \in [\ell]$.

Our protocol proceeds in five stages: in the first stage we let the parties commit to a key and exchange their inputs in an encrypted format (using a symmetric encryption scheme). In this way the inputs to all computations are well-defined already from this stage. We then let both parties garble $(1 + \epsilon)\ell$ copies of the circuit and, using *cut-and-choose*, verify that $\epsilon\ell$ of them are correct: this guarantees that even if one of the two parties has been actively corrupted, there are at most $O(\kappa)$ incorrect circuits among the unopened ones except with negligible probability. We then proceed to evaluate these circuits in both directions (as in the *dual-execution* protocol of Franklin and Mohassel [MF06]). Remember that Yao's protocol is "almost" actively secure against corrupt evaluators, which means that at this stage the corrupt party learns the correct output of the function (together with unforgeable output labels) in all positions, while the honest party learns outputs (with corresponding output labels) in at least $(\ell - O(\kappa))$ positions.[2] For the remaining $O(\kappa)$ positions the honest party might receive an incorrect output or no output at all. Therefore in the next stage, which we call the *filling-in stage*, we allow each party to ask for at most $O(\kappa)$ re-computations using an actively secure protocol (without disclosing in which positions), and we enforce that the computation is performed on the same inputs (remember that the inputs are provided in encrypted format). This computation also outputs MACs on the new results. Note that the malicious party cannot gain anything in this stage, since the corrupt party has already learned the correct output, and learning it once more does not leak any extra information. After this stage we are ensured that both parties have $\ell$ candidate outputs together with unforgeable certificates of their authenticity (either the output labels from the garbled circuits or the MACs from the do-overs). But still it might be that some of the outputs received by the honest party are incorrect and, therefore, different from the one received by the corrupt party. In the final step of the protocol we run a kind of *forge-and-lose* subprotocol, where both parties input all the received outputs to an actively secure computation. The computation finds the first position in which the outputs differ and recomputes the function for that index.

[2] As we cannot guarantee fairness nor termination in the two-party setting, the adversary can of course abort the protocol at this stage and prevent the honest party from learning any output at all.

Since the parties cannot lie about their outputs at this point (due to the unforgeable certificates), there is at most one party with the incorrect output, and that party is the honest one. Therefore, the other party must have cheated by garbling an incorrect circuit. To "punish" this party, we release the secret key of the malicious party to the honest party. This allows the honest party to decrypt all the encrypted inputs and compute all the correct results "in the clear".

To recap, here are the 5 stages of the protocol:

1. *(encrypted input)* Both parties exchange inputs. This is done by using a committed OT usinging some symmetric keys $(\sigma_A, \sigma_B) \in \{0,1\}^{O(\kappa)}$ as choice bits and then exchanging encrypted inputs $X_i^A = E(\sigma_A, x_i^A)$ and $X_i^B = E(\sigma_B, x_i^B)$.
2. *(cut-and-choose)* both parties garble $(1+\epsilon)\ell$ circuits which compute

$$(\sigma_A, X_A, \sigma_B, X_B) \mapsto f(D(\sigma_A, X_A), D(\sigma_B, X_B))$$

   and do cut-and-choose by checking $\epsilon\ell$ circuits each.
3. *(dual execution)* The parties evaluate each of the $\ell$ remaining circuits. Thanks to cut-and-choose, at most $O(\kappa)$ circuits are incorrect, which in particular means there are at most $O(\kappa)$ positions in which the honest party did not receive an output.
4. *(filling-in)* The parties run an actively secure protocol which recomputes the function in at most $O(\kappa)$ positions. Using Merkle-trees based commitments we can make sure that a) the functions are recomputed on the same inputs as before and that b) the input to this protocol (and its complexity) does not grow linearly with $\ell$. This protocol also outputs MACs for the recomputed values;
5. *(forge-and-lose)* At this point both parties have $\ell$ outputs, but if one party is dishonest some of the outputs might still be different. So now the parties run an actively secure protocol which a) finds the first position where the outputs are different and b) recomputes the function in that position, finds out which party cheated, and reveals the secret key of the corrupt party to the honest party, which can therefore decrypt all inputs and recompute the function in the clear. Since MACs are used, a corrupt party cannot input a wrong output (which would make the honest party to look corrupt).

Step 1 can be seen as a kind of *committed oblivious transfer* combined with an *oblivious transfer extension*, where we start with a "small" committed OT functionality for $O(\kappa)$ pair of messages which are then used to provide very long inputs to a garbled circuit based computation: if $a(\kappa)$ is the cost of evaluating a gate with an actively secure protocol and $q_1(\kappa)$ is some function describing the complexity of the circuit computing $O(\kappa)$ committed OTs on messages of length $O(\kappa)$, then the complexity of this step is $q_1(\kappa)a(\kappa)$. If we set e.g., $\epsilon = 1/4$ then the total complexity of Step 2 and 3 is $\leq 5\ell(|f| + q_2(\kappa))p(\kappa)$, where $q_2$ represents the complexity of the decryption circuit $D$. Then, the complexity of Step 4 is $\leq (\psi\kappa)(|f| + q_3(\kappa, \log \ell))a(\kappa)$ where $q_3$ is the complexity of verifying the Merkle-tree commitment and computing a MAC, and $\psi := \log_{1+\epsilon}(2)$ is a

constant picked to guarantee that the probability that there are more than $\psi\kappa$ bad circuits among the unchecked ones is less than $2^{-\kappa}$. Finally the complexity of Step 5 is $(\ell q_4(\kappa) + |f|)a(\kappa)$ where the $q_4$ factor represents the complexity of verifying the certificates. Since $a(\kappa) > p(\kappa)$ the total cost of the protocol is bounded by:

$$5\ell|f|p(\kappa) + (|f| + \ell)A(\kappa)$$

where $A(\kappa)$ collects all the terms which are independent of the circuit size or the number of computations. Now, when amortizing over $\ell$ executions, and assuming that $\ell$ is at least as large as $|f|$, we achieve the desired amortized complexity stated earlier in (1).

We can actually quantify the constant overhead over Yao's protocol even more precisely, by looking at the actual cost (in PRF calls) for garbling (or checking) vs. evaluating a gate in some of the most common garbling schemes (i.e., instead of upper bounding it with $p$). Let $g$ be the number of calls to a PRF (encryptions) performed during garbling/checking a gate and $e$ be the number of calls to a PRF (decryptions) performed during the evaluation of a gate. Then $(g+e)$ is the exact computational cost (per gate) in the passive version of Yao's protocol. In our protocol the exact cost is $(2+4\epsilon)g+2e$. In Yao's original garbling $(g, e) = (4, 4)$, in *point-and-permute* [BMR90] $(g, e) = (4, 1)$ and finally in the *half-gate* [ZRE15] $(g, e) = (4, 2)$ which means that when using $\epsilon = 1/4$ then the concrete overhead over Yao's passive protocol is between 2.5 and 2.8.[3]

We conclude by stressing that this work is of theoretical nature and therefore we have made no attempts in optimizing the concrete efficiency of any of the steps. On the contrary, since our protocol is already quite complex and involves several stages we have chosen at each turn simplicity of presentation over (concrete) efficiency. We leave it as an interesting open direction for future work to investigate whether the approach proposed in this paper might lead to practically efficiency.

## 2 Preliminaries and Notation

We review here the standard tools which are used in our protocol and their notation.

*Commitments.* We use a computationally binding and computationally hiding commitment scheme $\mathsf{Com}$ with commitment key $ck \leftarrow \mathsf{CGen}(1^\kappa)$, and we use an informative but slightly abusive notation: we write $\langle x \rangle \leftarrow \mathsf{Com}_{ck}(x, \mathsf{open}(x))$ where $\langle x \rangle$ is a commitment to the value $x$ using randomness $\mathsf{open}(x)$. In the proof we need the commitment to be extractable i.e., we need the simulator to be able to compute $x \leftarrow \mathsf{Ext}(\mathsf{td}, \langle x \rangle)$ using some trapdoor $\mathsf{td}$ associated to the commitment key $ck$.

[3] Decreasing $\epsilon$ decreases the overhead but increases the number of potentially unchecked bad circuits, therefore increasing the number of necessary do-overs in Stage 3.

*Merkle-tree Commitments.* We use Merkle-tree based commitments with the following interface: Given a string of elements from some alphabet $\boldsymbol{x} \in \Sigma^n$ it is possible to compute a short commitment by running $\mathsf{root} \leftarrow \mathsf{MT.C}(1^\kappa, \boldsymbol{x})$. It is possible to construct a proof for a give position $j \in [\ell]$ by computing $\pi \leftarrow \mathsf{MT.P}(\boldsymbol{x}, j)$ and the proof can be verified running $b \leftarrow \mathsf{MT.V}(\mathsf{root}, j, x', \pi)$ with $b \in \{\top, \bot\}$. We want that $b = \top$ when the prover is honest and $x' = x_j$ (i.e., *correctness*), that the proof is short $|\pi| = O(\kappa \log \ell)$ (*compactness*) and that no PPT adversary can produce a tuple $(\mathsf{root}, j, x, x', \pi, \pi')$ such that $x \neq x'$ and $\mathsf{MT.V}(\mathsf{root}, j, x, \pi) = \mathsf{MT.V}(\mathsf{root}, j, x', \pi') = \top$ (*computational binding*). We do not need the commitments to be hiding.

*Symmetric Encryption.* We use an IND-CPA symmetric encryption scheme $(\mathsf{SE.E}, \mathsf{SE.D})$ with key $\sigma \in \{0,1\}^{8\kappa}$. We use lowercase letters for plaintexts and uppercase letters for ciphertexts, so $X \leftarrow \mathsf{SE.E}(\sigma, x)$ and $x \leftarrow \mathsf{SE.D}(\sigma, X)$. We need the encryption scheme to be secure even if $\kappa$ bits of the secret key leak to the adversary (to counteract standard selective-failure attacks during the OT phase). This is done in the following way: we start by generating a uniformly random key $\kappa$ bit key $\sigma'$, which is then encoded into a $8\kappa$ bit long key $\sigma$ using the (randomized) encoding scheme $(\mathsf{enc}, \mathsf{dec})$ of Lindell and Pinkas [LP07] i.e., we compute $\sigma \leftarrow \mathsf{enc}(\sigma', r)$ with some randomness $r$. Now given any encryption scheme $E', D'$ which is IND-CPA secure using a $\kappa$-bit long key, we define $\mathsf{SE.E}, \mathsf{SE.D}$ to be $\mathsf{SE.E}(\sigma, m) = E'(\mathsf{dec}(\sigma), m)$ and $\mathsf{SE.D}(\sigma, c) = D'(\mathsf{dec}(\sigma), m)$.

*MAC Scheme.* We use an unforgeable message authentication code (MAC) $(\mathsf{MAC.Tag}, \mathsf{MAC.Ver})$ with key $\tau \in \{0,1\}^\kappa$ and the following interface: one can compute a tag on a message $x$ by computing $t \leftarrow \mathsf{MAC.Tag}(\tau, x)$ and the tag can be verified running $\mathsf{MAC.Ver}(t, \tau, x) \in \{\bot, \top\}$.

*Oblivious Transfer.* We use the following notation for transforming a random OT on a short, random messages (of length $\kappa$) into any number of OTs on messages of any lengths (with the same choice bits): we start with the sender knowing $\mathsf{sen} = \{r_0, r_1\}$ (a pair of random strings in $\{0,1\}^\kappa$) and the receiver knowing $\mathsf{rec} = \{r_\sigma\}$ (for a bit $\sigma \in \{0,1\}$). Then we write

$$\mathsf{tra}_j \leftarrow \mathsf{OTTransfer}(\mathsf{sen}, j, \{m_0, m_1\})$$

for the process of encrypting the pair of message $\{m_0, m_1\}$ using keys $r_0, r_1$ respectively (using an IND-CPA symmetric encryption scheme) and

$$m_\sigma \leftarrow \mathsf{OTRetrieve}(\mathsf{rec}, j, \mathsf{tra}_j)$$

for the process of recovering $m_{\sigma,j}$ from $\mathsf{tra}_j$. To ease the notation, we also allow a "vector" version of these OT commands i.e., if $\boldsymbol{e} = \{m_{i,0}, m_{i,1}\}_{i\in[n]}$ is a vector of $n$ pairs of messages and $\sigma \in \{0,1\}^n$ is a vector of $n$ bits then we write $\mathsf{rec} = \{r_{i,\sigma_i}\}_{i\in[n]}$, $\mathsf{sen} = \{r_{i,0}, r_{i,1}\}_{i\in[n]}$ for the information known to the receiver and sender respectively, $\mathsf{tra}_j \leftarrow \mathsf{OTTransfer}(\mathsf{sen}, j, e)$ for the process of encrypting each pair of messages and finally $M \leftarrow \mathsf{OTRetrieve}(\mathsf{rec}, j, \mathsf{tra}_j)$ with $M =$

$\{m_{i,\sigma_i}\}_{i\in[n]}$. (In the proof of security we also use $e \leftarrow \mathsf{OTRetrieve}(\mathsf{sen}, j, \mathsf{tra}_j)$ to denote the process of recovering all pairs of messages using the keys known to the sender).

*Garbled Circuits.* We use the generalization of the notation introduced by Bellare et al. [BHR12] already introduced in [JKO13, FNO15]: a garbling scheme is a tuple of algorithms

$$(\mathsf{GC.Gb}, \mathsf{GC.Ev}, \mathsf{GC.En}, \mathsf{GC.De}, \mathsf{GC.Ve})$$

where:

- $(\hat{f}, e, d) \leftarrow \mathsf{GC.Gb}(f; r)$ generates a garbled version $\hat{f}$ of the circuit $f : \{0,1\}^n \to \{0,1\}^n$ which has $n$ input bits and $n$ output bits. We make explicit the randomness $r$ used to garble since it will be used in the verification process. The $\mathsf{GC.Gb}$ function outputs the garbled version of the function $\hat{f}$, the encoding tables $e$ and the decoding tables for the output wires $d$;
- $\hat{x} \leftarrow \mathsf{GC.En}(e, x)$ outputs an encoding of $x$.
- $\hat{z} \leftarrow \mathsf{GC.Ev}(\hat{f}, \hat{x})$ outputs an encoded version of the output;
- $z' \leftarrow \mathsf{GC.De}(d, \hat{z})$ outputs the plaintext version of an encoded value $\hat{z}$ (or $\perp$ for an invalid encoding);
- $b \leftarrow \mathsf{GC.Ve}(f, r, \hat{f}, e, d)$ allows to verify if a given garbled circuit was garbled correctly and outputs $b \in \{\top, \perp\}$;

As usual we need the garbling scheme to be *projective* – i.e., both $(e, d)$ are vectors of pairs of strings – to be compatible with Yao's protocol. We need the garbling scheme to satisfy *privacy* and *authenticity* as defined in [BHR12]. We need the garbling scheme to be *verifiable* in the standard sense i.e., that an adversary cannot "open" a garbling $\hat{f}$ to any function different than $f$.

**Definition 1 (Correctness).** *We say that a garbling scheme enjoys* correctness *if for all $n = \mathrm{poly}(\kappa)$, $f : \{0,1\}^n \to \{0,1\}^n$ and all inputs $x \in \{0,1\}^n$:*

$$\Pr\left(f(x) \neq \mathsf{GC.De}(d, \mathsf{GC.Ev}(\hat{f}, \mathsf{GC.En}(e, x))) : (\hat{f}, e, d) \leftarrow \mathsf{GC.Gb}(1^\kappa, f)\right) = 0$$

*(the probability is taken over the random coins of all algorithms).*

**Definition 2 (Privacy).** *We way a garbling scheme enjoys* privacy *there exists a PPT simulator $\mathcal{S}$ such that the two following distributions are computationally indistinguishable:*

$$\{(\hat{f}, \mathsf{GC.En}(e, x), d) : (\hat{f}, e, d) \leftarrow \mathsf{GC.Gb}(1^\kappa, f)\}_x \approx \{\mathcal{S}(1^\kappa, f, f(x))\}_x$$

*for all $f, x$.*

**Definition 3 (Authenticity).** *We say a garbling scheme enjoys* authenticity *if for all PPT $\mathcal{A}$, for all $f, x \in \{0,1\}^n$*

$$\Pr\left(\mathsf{GC.De}(d, z^*) \neq \bot \wedge z^* \neq \mathsf{GC.Ev}(\hat{f}, \hat{x}) : \begin{array}{c}(\hat{f}, e, d) \leftarrow \mathsf{GC.Gb}(1^\kappa, f),\\ \hat{x} \leftarrow \mathsf{GC.En}(e, x),\\ z^* \leftarrow \mathcal{A}(\hat{f}, \hat{x})\end{array}\right)$$

*is negligible in $\kappa$.*

**Definition 4 (Circuit Verifiability).** *We say a garbling scheme enjoys* circuit verifiability *if for all PPT $\mathcal{A}$:*

$$\Pr\left(\mathsf{GC.Gb}(f_0; r_0) = \mathsf{GC.Gb}(f_1; r_1) : (f_0, r_0, f_1, r_1) \leftarrow \mathcal{A}(1^\kappa), f_0 \neq f_1\right)$$

*is negligible in $\kappa$.*

*Input Verification.* We also enhance the garbling scheme with two algorithms $(\mathsf{GC.TkG}, \mathsf{GC.TkV})$. The algorithm $\mathsf{GC.TkG}$ allows to generate some "tokens" $\mathsf{tk}$ from the input labels $e$. These tokens can be used with the $\mathsf{GC.TkV}$ algorithm to check whether an encoding of an input $\hat{x}$ is correct without leaking any information about the input $x$ itself. In a nutshell, we construct this from any projective garbling scheme in the following way: let $e = (K_0, K_1)$ be the encoding information of the original garbling scheme (for simplicity we assume a single input bit). Then we flip a random bit $r$ and let $\mathsf{tk} = (\langle K_r \rangle, \langle K_{1-r} \rangle)$ and $e' = ((K_0, \mathsf{open}(K_0)), (K_1, \mathsf{open}(K_1)))$ that is, we extend the input labels with some randomness and we compute two commitments, and permute them in a random order. Now given an encoding of an input $\hat{x} \leftarrow \mathsf{GC.En}(e', x)$ (using the extended labels i.e., $\hat{x} = (K_x, \mathsf{open}(K_x))$) it is possible to verify whether this is a correct encoding by running $\mathsf{GC.TkV}(\mathsf{tk}, \hat{x}) \in \{\top, \bot\}$. The algorithm simply parses $\hat{x} = (K^*, \mathsf{open}(K^*))$, computes $\langle K^* \rangle = \mathsf{Com}_{ck}(K^*, \mathsf{open}(K^*))$ and checks if $\langle K^* \rangle \in \mathsf{tk}$. We now that adding these tokens does not break the privacy property of the garbling scheme, and that adding the tokens we get the property that (for correctly generated circuits), if a (possibly malicious) encoding of an input passes the verification with the tokens, then the output of the evaluation will not be $\bot$.

**Token Generation:** Given any *projective* garbling scheme i.e., one where $e = \{(K_0^i, K_1^i)\}_{i \in [n]}$, we construct a new garbling scheme with *verifiable input* in the following way: First we define the new encoding information $e'$ to be

$$e' = \{(K_0^i, \mathsf{open}(K_0^i)), (K_1^i, \mathsf{open}(K_1^i))\}_{i \in [n]}$$

and then we compute tokens $\mathsf{tk} \leftarrow \mathsf{GC.TkG}(e')$ by sampling random bits $r_1, \ldots, r_n$ and outputting

$$\mathsf{tk} = \{\langle K_{r_i}^i \rangle, \langle K_{1-r_i}^i \rangle\}_{i \in [n]}$$

With $\langle K_b^i \rangle = \mathsf{Com}_{ck}(K_b^i, \mathsf{open}(K_b^i))$ for all $b \in \{0, 1\}, i \in [n]$.

**Input Verification:** $b \leftarrow \mathsf{GC.TkV}(\mathsf{tk}, \hat{x})$ is a deterministic algorithm that parses

$$\hat{x} = \{K^i, \mathsf{open}(K^i)\}_{i\in[n]}$$

computes $\langle K^i \rangle = \mathsf{Com}_{ck}(K^i, \mathsf{open}(K^i))$, and outputs $\perp$ if there exists an $i$ such that $\langle K^i \rangle \notin \mathsf{tk}$.

We define the following properties:

**Definition 5 (Token Privacy).** *We way a garbling scheme enjoys* token privacy *if there exists a PPT simulator $\mathcal{S}$ such that the two following distributions are computationally indistinguishable:*

$$\left\{ (\hat{f}, \mathsf{GC.En}(e, x), d, \mathsf{tk}) : \begin{array}{c} (\hat{f}, e, d) \leftarrow \mathsf{GC.Gb}(1^\kappa, f), \\ \mathsf{tk} \leftarrow \mathsf{GC.TkG}(e) \end{array} \right\}_x \approx \{\mathcal{S}(1^\kappa, f, f(x))\}_x$$

**Definition 6 (Input Verifiability).** *We way a garbling scheme enjoys* input verifiability *if for all PPT $\mathcal{A}$ the following probability*

$$\Pr\left( \mathsf{GC.De}(d, \mathsf{GC.Ev}(\hat{f}, \hat{x}^*)) = \perp \wedge \mathsf{GC.TkV}(\mathsf{tk}, \hat{x}^*) = \top : \begin{array}{c} (\hat{f}, e, d) \leftarrow \mathsf{GC.Gb}(1^\kappa, f), \\ \mathsf{tk} \leftarrow \mathsf{GC.TkG}(e), \\ \hat{x}^* \leftarrow \mathcal{A}(1^\kappa, \hat{f}, e), \end{array} \right)$$

*is negligible in $\kappa$.*

The proof that our construction satisfies the above requirements is straightforward.

**Lemma 1.** *Any projective garbling scheme can be enhanced to achieve token privacy and input verifiability using computationally hiding and computationally binding commitments as described above.*

*Proof.* For *token privacy*, we simply run the simulator guaranteed by the *privacy* property of the underlying garbling scheme. In addition, our simulator needs to output $\mathsf{tk}$, a vector of pair of commitments. The simulator does so by parsing the encoded input $\hat{x}$ (provided by the privacy simulator) into $\hat{x} = \{K^i\}_{i\in[n]}$, chooses random values $\mathsf{open}(K^i)$, computes commitments $\langle K^i \rangle = Com(K^i, \mathsf{open}(K^i))$. The simulator also constructs $n$ commitments $\langle 0 \rangle$ (using independent randomness), and constructs $\mathsf{tk}$ as $n$ pairs of commitments, where each pair is $(\langle K^i \rangle, \langle 0 \rangle)$ if $r_i = 0$ or $(\langle 0 \rangle, \langle K^i \rangle)$ otherwise. Any adversary that can distinguish between the two distributions in the token privacy property can be trivially reduced to an adversary for either the computationally hiding property of the commitment scheme or the privacy property of the underlying scheme.

For the *input verifiability* property, let $\hat{x}^* = \{K^i, \mathsf{open}(K^i)\}_{i\in[n]}$ be the output of the adversary and let $e = \{(K_0^i, \mathsf{open}(K_0^i)), (K_1^i, \mathsf{open}(K_1^i)))\}_{i\in[n]}$. Now if $\mathsf{GC.TkV}(\mathsf{tk}, \hat{x}^*) = \top$, it must be the case that $K^i = K_b^i$ for some $b$ or the adversary can be used to break the binding property of the commitment scheme. But then, the property follows from the correctness of the underlying garbling scheme.

*Sub-functionalities.* In some stages of our protocol we let the parties run *any* actively-secure two-party computation protocol to implement a desired functionality. In the protocol description we only describe *what* the functionality should do and not *how* the functionality is implemented (in the proof we will make use of the UC composition theorem to replace these subprotocols with hybrid functionalities under the control of the simulator). In particular we describe the private input of both parties, the private output of each party and the computation performed by the functionality. We also describe the *public input* of some functionalities. These are values which are defined previously in the protocol which can be imagined as values which are given as input by both parties (and the functionality aborts if they are different).

## 3 Our Protocol

We are now ready to present our protocol, which we like to call *cross and clean*. As our protocol is quite complex, we split its presentation in five stages, which are described in Figures 1, 2, 3, 4 and 5 respectively.

**Stage 1: Providing Inputs:**

1. A, B run an actively secure protocol which securely implements the following functionality:

**Input:** none.
**Computation:** The functionality runs the following code:

1. Generate commitment keys $ck^A \leftarrow \mathsf{CGen}(1^\kappa)$, $ck^B \leftarrow \mathsf{CGen}(1^\kappa)$;
2. Sample random $\sigma^A, \sigma^B \in \{0,1\}^{8\kappa}$;
3. Compute $\langle \sigma^A \rangle \leftarrow \mathsf{Com}_{ck^A}(\sigma^A, \mathsf{open}(\sigma^A))$;
4. Compute $\langle \sigma^B \rangle \leftarrow \mathsf{Com}_{ck^B}(\sigma^B, \mathsf{open}(\sigma^B))$;
5. Sample random $\mathsf{sen}^A = \{r^A_{0,i}, r^A_{0,i}\}_{i\in[8k]}$ with each $r^A_{b,i} \in \{0,1\}^\kappa$;
6. Sample random $\mathsf{sen}^B = \{r^B_{0,i}, r^B_{0,i}\}_{i\in[8k]}$ with each $r^B_{b,i} \in \{0,1\}^\kappa$;
7. Define $\mathsf{rec}^B = \{r^A_{\sigma^A[i],i}\}_{i\in[8k]}$;
8. Define $\mathsf{rec}^A = \{r^B_{\sigma^B[i],i}\}_{i\in[8k]}$;

**A's output:** The functionality sends as private output to A:

1. $\sigma^A$, $\mathsf{open}(\sigma^A)$, $\langle \sigma^B \rangle$;
2. $\mathsf{sen}^A, \mathsf{rec}^A$;

**B's output:** The functionality sends as private output to B:

1. $\sigma^B$, $\mathsf{open}(\sigma^B)$, $\langle \sigma^A \rangle$;
2. $\mathsf{sen}^B, \mathsf{rec}^B$;

2. For all $i \in [\ell]$: A sends $X^A_i = \mathsf{SE.E}(\sigma^A, x^A_i)$ to B;
3. For all $i \in [\ell]$: B sends $X^B_i = \mathsf{SE.E}(\sigma^B, x^B_i)$ to A;

**Figure 1.** Stage 1: Providing Inputs

**Stage 2: Cut-n-Choose:**

1. A garbles $(1+\epsilon)\ell$ times (with independent randomness)[a]:

$$(\hat{f}_i^A, e_i^A, d_i^A) \leftarrow \mathsf{GC.Gb}(f_i^A; r_i^A)$$

   where $f_i^A$ is the circuit that computes

$$y_i^B = f(\mathsf{SE.D}(\sigma^A, X_i^A), \mathsf{SE.D}(\sigma^B, X_i^B))$$

2. A computes $\mathsf{tk}_i^A \leftarrow \mathsf{GC.TkG}(e_i^A)$ and $\langle d_i^A \rangle \leftarrow \mathsf{Com}_{ck^A}(d_i^A, \mathsf{open}(d_i^A))$;
3. A sends $(\hat{f}_i^A, \mathsf{tk}_i^A, \langle d_i^A \rangle)$ to B;
4. A,B perform an actively secure coin flip to determine the random subset

$$\mathrm{CC} \subset \{1, \ldots, (1+\epsilon)\ell\}$$

   of size $\epsilon\ell$ of circuits to be checked;
5. A sends $(r_i^A, e_i^A, d_i^A, \mathsf{open}(d_i^A))$ to B;
6. B aborts if

$$\exists i \in \mathrm{CC}: \quad \begin{array}{c} \mathsf{GC.Ve}(f_i^A, e_i^A, d_i^A, r_i^A, \hat{f}_i^A) = \bot \text{ or} \\ \mathsf{tk}_i^A \neq \mathsf{GC.TkG}(e_i^A) \text{ or} \\ \langle d_i^A \rangle \neq \mathsf{Com}_{ck^A}(d_i^A, \mathsf{open}(d_i^A)) \end{array}$$

7. A and B reindex the unopened circuits $1, \ldots, \ell$.
8. A,B repeat with reversed roles;

[a] For the sake of notation, we implicitly assume that $\epsilon\ell$ is an integer.

**Figure 2.** Stage 2: Cut-n-Choose

**Theorem 1.** *The protocol described in Figures 1, 2, 3, 4 and 5 securely evaluates $\ell$ copies of $f$ in the presence of active adversaries.*

*Proof.* Thanks to the UC composition theorem it is sufficient to prove security of the protocol where we replace all actively secure subprotocols in the protocol (the *committed OT*, *coin-flip*, *filling-in* and *forge-and-lose* subprotocols respectively in Stage 1, 2, 4, 5) with ideal functionalities controlled by the simulator. Since the protocol is completely symmetric for A and B, we will assume in the proof that A is corrupt and B is honest.

As usual we make a proof by hybrids. We describe the simulator strategy along the way, by making progressive changes from the real protocol towards the simulator strategy, arguing for indistinguishability after every change. We describe the final simulator strategy in Figure 6.

*Hybrid 0.* We start in dream version of the simulation, where we assume the simulator is given the real inputs $x_i^B$ of the honest party $B$.

Then the simulator then simulates simply by running the real protocol with the adversary controlling $A$ and the simulator running $B$ and the hybrid ideal functionalities. If $B$ aborts in the protocol before receiving encrypted inputs

**Stage 3: Run Computation $i$:**

1. $A$ computes encoded versions of the inputs $\hat{X}_i^A, \hat{X}_i^B, \hat{\sigma}_i^A$ i.e.,
   (a) $\hat{X}_i^A \leftarrow \mathsf{GC.En}(e_i^A, X_i^A)$,
   (b) $\hat{X}_i^B \leftarrow \mathsf{GC.En}(e_i^A, X_i^B)$,
   (c) $\hat{\sigma}_i^A \leftarrow \mathsf{GC.En}(e_i^A, \sigma^A)$,
   and sends them to B;
2. A runs $\mathsf{tra}_i^A \leftarrow \mathsf{OTTransfer}(\mathsf{sen}^A, i, e_i^A)$ and sends $\mathsf{tra}_i^A$ to B;
3. B runs $\hat{\sigma}_i^B \leftarrow \mathsf{OTRetrieve}(\mathsf{rec}^B, i, \mathsf{tra}_i^A)$;
4. B aborts if $\mathsf{GC.TkV}(\mathsf{tk}_i^A, (\hat{\sigma}_i^A, \hat{X}_i^A, \hat{\sigma}_i^B, \hat{X}_i^B)) = \perp$;
5. B evaluates $\hat{y}_i^B = \mathsf{GC.Ev}(\hat{f}_i^A, \hat{X}_i^A, \hat{X}_i^B, \hat{\sigma}_i^A, \hat{\sigma}_i^B)$;
6. B computes $\langle \hat{y}_i^B \rangle \leftarrow \mathsf{Com}_{ck^B}(\hat{y}_i^B, \mathsf{open}(\hat{y}_i^B))$
7. B sends $\langle \hat{y}_i^B \rangle$ to A;
8. A sends $d_i^A, \mathsf{open}(d_i^A)$ to B; B aborts if $\mathsf{Com}_{ck}(d_i^A, \mathsf{open}(d_i^A)) \neq \langle d_i^A \rangle$
9. Bob computes $y_i^B \leftarrow \mathsf{GC.De}(d_i^A, \hat{y}_i^B)$;
10. If $y_i^B = \perp$ add $i$ to $I^B$;
11. If $|I^B| > \psi\kappa$, then abort the protocol.
12. A,B repeat with reversed roles;

**Figure 3.** Stage 3: Run Computation $i$

from $A$, instruct the ideal functionality to abort on behalf of the corrupted $A$. Otherwise the simulator extracts $\{x_i^A \leftarrow \mathsf{SE.D}(\sigma^A, X_i^A)\}_{i\in[\ell]}$ and input these values to the ideal functionality. This allows the simulator to learn all the real outputs $\{y_i\}_{i\in[\ell]}$. If $B$ aborts in the protocol after receiving encrypted inputs, instruct the ideal functionality to abort on behalf of the corrupted $A$. Otherwise, let $\{y_i'\}_{i\in[\ell]}$ be the outputs of the protocol. If $\{y_i\}_{i\in[\ell]} \neq \{y_i'\}_{i\in[\ell]}$, the simulator aborts the simulation. Clearly this first hybrid is perfectly indistinguishable from the real protocol execution as long as $\{y_i\}_{i\in[\ell]} = \{y_i'\}_{i\in[\ell]}$. Therefore hybrid 0 is computational indistinguishable from the real protocol execution if the protocol has correctness except with negligible probability. We argue correctness of this at the end of the proof.

*Hybrid 1.* Here we change the inner working of the *committed OT* functionality (Stage 1): from now on the simulator sends A a commitment to 0 instead of $\sigma^B$. Any adversary that can distinguish after this change can be used to break the computationally hiding property of the commitment scheme.

*Hybrid 2.* Here we replace the commitment sent to A in Stage 4 (*filling in phase*) to be a commitment to 0 instead of the MAC key $\tau^B$. Any adversary that can distinguish after this change can be used to break the hiding property of the commitment.

*Hybrid 3.* Here we replace the abort condition (Step 1.a) in the functionalities for *filling-in* in Stage 4. Let $\sigma^A$ be the values received by A from the *committed*

**Stage 4: Actively Secure Filling-in:**

1. Define $V_i = (X_i^A, X_i^B)$. Both A and B compute $\mathsf{root} = \mathsf{MT.C}(V_1, \ldots, V_\ell)$;
2. Run an actively secure protocol which securely implements the following functionality:

**Public Input:** The functionality takes public inputs $\mathsf{root}, \langle \sigma^A \rangle, \langle \sigma^B \rangle$.

**A's input** The functionality takes as private input from A:

1. $\sigma^A$, $\mathsf{open}(\sigma^A)$,
2. the set $I^A$ with $|I^A| \leq \psi\kappa$ and
3. $(V_i, \pi_i \leftarrow \mathsf{MT.P}((V_1, \ldots, V_\ell), i)$ for all $i \in I^A$.

**B's input** The functionality takes as private input from B:

1. $\sigma^B$, $\mathsf{open}(\sigma^B)$,
2. the set $I^B$ with $|I^B| \leq \psi\kappa$ and
3. $(V_i, \pi_i \leftarrow \mathsf{MT.P}((V_1, \ldots, V_\ell), i)$ for all $i \in I^B$.

**Computation:** The functionality runs the following code:

1. Abort if any of the following is true:
   (a) $\langle \sigma^A \rangle \neq \mathsf{Com}_{ck^A}(\sigma^A, \mathsf{open}(\sigma^A))$,
   (b) $\langle \sigma^B \rangle \neq \mathsf{Com}_{ck^B}(\sigma^B, \mathsf{open}(\sigma^B))$,
   (c) $\exists i : I^A \cup I^B : \mathsf{MT.V}(\mathsf{root}, i, V_i, \pi_i) = \perp$;
2. If all checks pass
   (a) Sample random $\tau^A, \tau^B \in \{0,1\}^\kappa$;
   (b) Compute $\langle \tau^A \rangle \leftarrow \mathsf{Com}_{ck^A}(\tau^A, \mathsf{open}(\tau^A))$;
   (c) Compute $\langle \tau^B \rangle \leftarrow \mathsf{Com}_{ck^B}(\tau^B, \mathsf{open}(\tau^B))$;
3. For all $i \in I^A \cup I^B$:
   (a) Parse $V_i = (X_i^A, X_i^B)$ and compute
   (b) $x_i^A \leftarrow \mathsf{Dec}(\sigma^A, X_i^A)$;
   (c) $x_i^B \leftarrow \mathsf{Dec}(\sigma^B, X_i^B)$;
   (d) $y_i = f(x_i^A, x_i^B)$;
   (e) $t_i^A = \mathsf{MAC.Tag}(\tau^B, y_i)$ if $i \in I^A$ or $t_i^B = \mathsf{MAC.Tag}(\tau^A, y_i)$ otherwise;

**A's output:** The functionality sends as private output to A:

1. $\tau^A$, $\mathsf{open}(\tau^A)$, $\langle \tau^B \rangle$;
2. For all $i \in I^A : (y_i^A, t_i^A)$

**B's output:** The functionality sends as private output to B:

1. $\tau^B$, $\mathsf{open}(\tau^B)$, $\langle \tau^A \rangle$;
2. For all $i \in I^B : (y_i^B, t_i^B)$;

**Figure 4.** Stage 4: Actively Secure Filling-in

*OT* functionality in Stage 1 and $\sigma^*$ be the values input by A to the *filling-in* functionality in Stage 4. From now on we always abort if $\sigma^* \neq \sigma^A$ (even if what A inputs is a valid opening of the commitment $\langle \sigma^A \rangle$). Any adversary that can distinguish after this change can be used to break the computationally binding property of the commitment scheme. (Note that from now on we have the guarantee that the output of *filling-in* to the honest party B can only be the real output $y_i$.)

**Stage 5: Forge-and-Lose:**

1. A defines $t_i^A = \perp$ for all $i \notin I^A$;
2. B defines $t_i^B = \perp$ for all $i \notin I^B$;
3. A and B run a protocol implementing the following functionality with active security:

**Public Input:**

1. $\langle \sigma^A \rangle, \langle \sigma^B \rangle, \langle \tau^A \rangle, \langle \tau^B \rangle$
2. for all $i \in [\ell] : (d_i^A, d_i^B, X_i^A, X_i^B, \langle \hat{y}_i^A \rangle, \langle \hat{y}_i^B \rangle)$;

**A's input** The functionality takes as private input from A:

1. $\sigma^A$, $\mathsf{open}(\sigma^A)$,
2. $\tau^A$, $\mathsf{open}(\tau^A)$;
3. For all $i \in [\ell] : (y_i^A, \hat{y}_i^A, \mathsf{open}(\hat{y}_i^A), t_i^A)$;

**B's input** The functionality takes as private input from B:

1. $\sigma^B$, $\mathsf{open}(\sigma^B)$,
2. $\tau^B$, $\mathsf{open}(\tau^B)$;
3. For all $i \in [\ell] : (y_i^B, \hat{y}_i^B, \mathsf{open}(\hat{y}_i^B), t_i^B)$;

**Computation:** The functionality:

1. Abort if any of the following is true:
   (a) $\langle \sigma^A \rangle \neq \mathsf{Com}_{ck^A}(\sigma^A, \mathsf{open}(\sigma^A))$,
   (b) $\langle \sigma^B \rangle \neq \mathsf{Com}_{ck^B}(\sigma^B, \mathsf{open}(\sigma^B))$,
   (c) $\langle \tau^A \rangle \neq \mathsf{Com}_{ck^A}(\tau^A, \mathsf{open}(\tau^A))$,
   (d) $\langle \tau^B \rangle \neq \mathsf{Com}_{ck^B}(\tau^B, \mathsf{open}(\tau^B))$,
   (e) $\exists i : \langle \hat{y}_i^A \rangle \neq \mathsf{Com}_{ck^A}(\hat{y}_i^A, \mathsf{open}(\hat{y}_i^A))$,
   (f) $\exists i : \langle \hat{y}_i^B \rangle \neq \mathsf{Com}_{ck^B}(\hat{y}_i^B, \mathsf{open}(\hat{y}_i^B))$,
   (g) $\exists i : y_i^B \neq \mathsf{GC.De}(d_i^A, \hat{y}_i^B) \wedge \mathsf{MAC.Ver}(t_i^B, \tau^A, y_i^B) = \perp$;
   (h) $\exists i : y_i^A \neq \mathsf{GC.De}(d_i^B, \hat{y}_i^A) \wedge \mathsf{MAC.Ver}(t_i^A, \tau^B, y_i^A) = \perp$;
2. If all checks pass
   (a) Find $i : y_i^A \neq y_i^B$;
   (b) If no such $i$ exists conclude that "no-one cheated";
   (c) Else compute $y_i = f(x_i^A, x_i^B)$;
   (d) Conclude that "Alice cheated" if: $y_i^B \neq y_i$;
   (e) Conclude that "Bob cheated" if: $y_i^A \neq y_i$;

**Output:** If

1. 'Alice cheated" : A receives $\perp$, B receives $\sigma^A$;
2. 'Bob cheated" : A receives $\sigma^B$, B receives $\perp$;
3. else ('No-one cheated"): A receives $\perp$, B receives $\perp$;

4. If A (resp. B) receives an output $\sigma^B \neq \perp$, A computes $x_i^B \leftarrow \mathsf{SE.D}(\sigma^B, X_i^B)$ for all $i \in [\ell]$ and outputs $y_i^A = f(x_i^A, x_i^B)$;

**Figure 5.** Stage 5: Forge-and-Lose

*Hybrid 4.* Here we replace the abort condition (Step 1.a and 1.c) in **Computation** the functionality for *forge-and-lose* in Step 5 in a similar way as in the previous hybrid, namely: let $\sigma^*, \tau^*$ be the keys input by A, then in this hybrid we

**Simulator Strategy:**

**Stage 1:** 1. Simulate the *committed OT* functionality by sending A a random key $(\sigma^A, \mathsf{open}(\sigma^A))$; sending A a commitment $\langle 0 \rangle$ (instead of $\langle \sigma^B \rangle$); sending A random strings $\mathsf{sen}^A, \mathsf{rec}^A$; learning the trapdoor $\mathsf{td}^A$ corresponding to $ck^A$;

2. Receive $\{X_i^A\}_{i\in[\ell]}$, decrypt $x_i^A \leftarrow \mathsf{SE.D}(\sigma^A, X_i^A)$ and input these values to the ideal functionality; receive $\{y_i\}_{i\in[\ell]}$ from the ideal functionality;
3. Send A $\{X_i^B = \mathsf{SE.E}(\sigma^B, 0)\}_{i\in[\ell]}$;

**Stage 2.A:** In Stage 2.A the simulator (when receiving garbled circuits from A):

1-7. Receive garbled circuits; Sample a random CC and simulate the *coin-flip* functionality; Verify the circuits with $i \in$ CC and abort as an honest B would;

**Stage 2.B** In Stage 2.B the simulator (when sending garbled circuits to A):

1-7. Sample a random set CC. Construct honest garbling of the function for all $i \in$ CC; For all $i \notin$ CC, run the simulator of the garbling scheme to receive the garbled circuit $\hat{f}_i^B$, inputs $\hat{\sigma}_i^A, \hat{\sigma}_i^B, \hat{X}_i^A, \hat{X}_i^B$, decoding tables $d_i^A$ and tokens $\mathsf{tk}_i^B$. Force the *coin-flip* subprotocol to output CC;

**Stage 3.A:** In Stage 3 the simulator (when acting as circuit evaluator):

1. Receive garbled inputs;
2-4. Fully decrypt $\mathsf{tra}_i^A$, compute the set $\mathcal{L}$ and abort according to the strategy as described during Hybrid 11;
5-7. Compute and send A a commitment to $\langle 0 \rangle$;
8-10. Abort if $\langle d_i^A \rangle$ is not opened correctly;
11. There is no abort if $|I^B| > \psi\kappa$ as the simulator cannot compute $I^B$.

**Stage 3.B:** In Stage 3 the simulator (when acting as circuit constructor):

1. Send garbled inputs (from the garbling scheme simulator);
2-4. Prepare and send $\mathsf{tra}_i^B$ to A as described during Hybrid 6 (i.e., only send labels corresponding to the bits of $\sigma^A$, and 0 in all other positions); Fully decrypt $\mathsf{tra}_i$, compute the set $\mathcal{L}$ and abort according to the strategy as described during Hybrid 11;
5-7. Receive from A a commitment $\langle \hat{y}_i^B \rangle$ and extract $\hat{y}_i^{**} \leftarrow \mathsf{Ext}(\mathsf{td}^A, \langle \hat{y}_i^A \rangle)$ and abort if $\hat{y}_i^{**}$ breaks *authenticity*;
8-10. Open $\langle d_i^B \rangle$;

**Stage 4:** In Stage 4 the simulator

1. computes $\mathsf{root}$ as an honest B would;
2. Simulate the *filling-in* functionality by aborting if (on top of the original conditions) $\sigma^* \neq \sigma^A$ or $V_i^* \neq V_i$ where $\sigma^*, \{V_i^*\}_{i\in I^A}$ are the value sent by the adversary to the functionality; If the simulator does not abort, it sends A a random MAC key and commitment opening $(\tau^A, \mathsf{open}(\tau^A))$ and a commitment $\langle 0 \rangle$ (instead of $\tau^B$);

**Stage 5:** In Stage 5 the simulator

1-3. simulates the *forge-and-lose* functionality by aborting if (on top of the original conditions) $\sigma^* \neq \sigma^A$, $\tau^* \neq \tau^A$, $\hat{y}_i^{**} \neq \hat{y}_i^*$, $y_i^* \neq y_i$ where $y_i^*, \sigma^*, \tau^*, \hat{y}_i^*$ are the value sent by the adversary to the functionality;

**Figure 6.** Simulator Strategy

abort if $\sigma^A \neq \sigma^*$ or $\tau^A \neq \tau^*$ (even if A inputs proper commitment openings). An adversary distinguishing after this change can again be used to break the computationally binding property of the commitment scheme.

*Hybrid 5.* Here we replace the last abort condition (Step 1.c) in the functionality for *filling-in* in Stage 4. For each $i \in I^A$ let $V_i^*$ be the value input by A to this computation. From now on we always abort if $V_i^* \neq V_i$ (even if the $\mathsf{MT.V}$ algorithm accepts the proof $\pi_i$). An adversary distinguishing after this change can be used to break the computationally binding property of the Merkle-tree commitment. (Note that at this point, by definition, the output of *filling-in* to A cannot be different from $y_i$).

*Hybrid 6.* Here we change the distribution of $\mathsf{tra}_i^B$ (the value sent to A in Step 2 of Stage 3 by B): instead of computing $\mathsf{tra}_i^B = \mathsf{OTTransfer}(\mathsf{sen}^B, i, e_i^B)$, we compute $\mathsf{tra}_i^B = \mathsf{OTTransfer}(\mathsf{sen}^B, i, e_i^*)$ where $e_i^* = \{K_0^i, K_1^i\}$ is defined as follows: let $\hat{\sigma}_i^A \leftarrow \mathsf{GC.En}(e_i^B, \sigma^A)$ and parse $\hat{\sigma}_i^A = \{K^i\}$, then we set

$$K^i_{\sigma^A[i]} = K^i \text{ and } K^i_{1-\sigma^A[i]} = 0 \ .$$

That is, we set all labels not corresponding to the bits of $\sigma^A$ to 0 in the other positions. Since A only has access to the keys $\mathsf{rec}^A$ corresponding to the bits of $\sigma^A$, we can use an adversary that distinguishes after this change to break the IND-CPA of underlying symmetric encryption scheme.

*Hybrid 7.* Here we change the distribution of the garbled circuits and garbled inputs sent to A during Stage 2 and 3 for all $i \notin \mathrm{CC}$ (since the simulator is controlling the coin flip functionality, this set is known to the simulator from the beginning), by running the simulator (which is guaranteed to exist thanks to the *token privacy* property of the garbling scheme on input the function $f$ and the output $y_i$. (For completeness token privacy is defined in Definition 5 in the appendix.) The simulator provides us with garbled versions of all inputs including $\hat{\sigma}_i^A$, as well as $\mathsf{tk}_i^B$, $d_i^B$ and $\hat{f}_i^B$ which can now replace the values sent to A in Step 3 of Stage 2 and Steps 1, 2 and 8 of Stage 3. Any adversary distinguishing after this step can be used to break the *token privacy* of the underlying garbling scheme. (Note that the simulator can also, running $\mathsf{GC.Ev}$ on the garbled circuit and the garbled inputs, compute the garbled output $\hat{y}_i^A$ which is needed in the next steps.)

*Hybrid 8.* Here we replace the abort condition (Step 1.e) in the functionality *forge-and-lose* in Stage 5. For all $i$ we let the simulator compute $\hat{y}_i^* \leftarrow \mathsf{Ext}(\mathsf{td}^A, \langle \hat{y}_i^* \rangle)$ using the trapdoor $\mathsf{td}^A$ (which the simulator learns as it controls the *committed-OT* subprotocol) and the commitments $\langle \hat{y}_i^* \rangle$ received during Step 6 of Stage 3. Now let $\hat{y}_i^{**}$ be the value input by A to the *forge-and-lose* functionality. The simulator now aborts if $\hat{y}_i^* \neq \hat{y}_i^{**}$ even if A provides a valid commitment opening. Any adversary distinguishing after this step can be used to break the binding property of the commitment scheme.

*Hybrid 9.* Here we change the last aborting condition (Step 1.h) in the functionality *forge-and-lose* in Stage 5. Let $(y_i^*, \hat{y}_i^*, t_i^*)$ be the value input by the adversary and let $(y_i, \hat{y}_i^A, t_i^A)$ be the values computed by the simulator in the previous hybrids. From now on, instead of aborting if

$$\exists i : y_i^* \neq \mathsf{GC.De}(d_i^B, \hat{y}_i^*) \wedge \mathsf{MAC.Ver}(t_i^*, \tau^B, y_i^*) = \perp$$

we abort if

$$\exists i : (y_i^* \neq \mathsf{GC.De}(d_i^B, \hat{y}_i^*) \wedge \mathsf{MAC.Ver}(t_i^*, \tau^B, y_i^*) = \perp) \vee (y_i^* \neq y_i) \ .$$

Any adversary that can distinguish after this change can be used to break unforgeability of the MAC scheme (note that the simulator at this point does not need to know $\tau^B$ since it has been replaced by 0 in the commitment that A receives at the end of the *filling-in* stage, and we can therefore successfully run the reduction) or to break the *authenticity* property of the garbling scheme (note that we have already made sure that value $\hat{y}_i^*$ input by A here is the same as the one he commits to in Step 7 of Stage 3 and – since the simulator can extract the value in the commitment using the trapdoor – the reduction can already break the authenticity property *before* having to send $d_i^B$ or the opening of the commitment to A in Step 8 of Stage 3). Note that after this change we are ensured (by definition) that A will never receive $\sigma^B$ as a result of running the *forge-and-lose* subprotocol.

*Hybrid 10.* Here we change the distribution of the commitment that the simulator sends to A in Step 6 of Stage 3, from being a commitment to $\hat{y}_i^B$ to being a commitment to 0. An adversary that distinguishes after this change can be used to break the hiding property of the commitment scheme.

*Hybrid 11.* Here we let the simulator fully decrypt the transfer message $\mathsf{tra}_i$ from A in Step 3 of Stage 3. That is, instead of running $\hat{\sigma}_i^B \leftarrow \mathsf{OTRetrieve}(\mathsf{rec}^B, i, \mathsf{tra}_i^A)$ the simulator extracts

$$e_i^* \leftarrow \mathsf{OTRetrieve}(\mathsf{sen}^A, i, \mathsf{tra}_i^A)$$

and constructs the set

$$\mathcal{L}_i \subset [8\kappa] \times \{0,1\}$$

as follows. Parse

$$\mathsf{tk}_i^A = \{\langle A_j \rangle, \langle B_j \rangle\}_{j \in [8\kappa]}$$

and

$$e_i^* = \{(K_{j,0}, \mathsf{open}(K_{j,0})), (K_{j,1}, \mathsf{open}(K_{j,1}))\}_{j \in [8\kappa]} \ .$$

We add $(j, b)$ to $\mathcal{L}_i$ if

$$\mathsf{Com}_{ck}(K_{j,b}, \mathsf{open}(K_{j,b})) \notin \{\langle A_j \rangle, \langle B_j \rangle\} \ .$$

We then compute

$$\mathcal{L} = \cup_{i \in [\ell]} \mathcal{L}_i \ .$$

The set $\mathcal{L}$ represents all the positions in all the OT transfers in which A "cheated" i.e., where A sent some value which is not consistent with the tokens $\mathsf{tk}_i^A$. Since an honest B uses the same input bits $\sigma^B$ in all transfers, we only count each combination of position $j$ and bit $b$ once. In other words, for each index $j$ A has three strategies:

1. Input the right values (i.e., values that make $\mathsf{GC.TkV}$ accept) for both $b \in \{0,1\}$ (for all $i \in [\ell]$): in this case neither $(j,b) \notin \mathcal{L}$ for both $b \in \{0,1\}$;
2. Input the right value for a single $b \in \{0,1\}$ (for all $i \in [\ell]$)) and at least a wrong value for $1-b$ for some $i^* \in [\ell]$: in this case $(j,1-b) \in \mathcal{L}$;
3. Input the wrong value for both $b \in \{0,1\}$ (potentially for different $i^* \in [\ell]$): in this case both $(j,0) \in \mathcal{L}$ and $(j,1) \in \mathcal{L}$;

We now replace the abort condition in Step 4 of Stage 3 to the following: the simulator aborts with probability 1 if $\exists\ j$ such that both $(j,0) \in \mathcal{L}$ and $(j,1) \in \mathcal{L}$ (this is consistent with what B would do in the real protocol, since in this case B will detect the wrong labels regardless of the value of $\sigma^B[j]$). Otherwise, the simulator aborts with probability $1 - 2^{|\mathcal{L}|}$ (this is consistent with what B would do in the real protocol, since in this case B detects the wrong labels only if $\sigma^B[j] = b$ for $(j,b) \in \mathcal{L}$ – note on the other hand that if $(j,b) \in \mathcal{L}$ and B does not abort then the corrupt A learns that the value of $\sigma^B[j] \neq b$, and we will take care of this in a moment). At the same time we change the distribution of the encryptions $X_i^B$ sent by B to A in Step 3 of Stage 1 to be all encryptions of 0. Any adversary that can distinguish after this change can be used to break the IND-CPA security of $(\mathsf{SE.E}, \mathsf{SE.D})$. Remember that we required $(\mathsf{SE.E}, \mathsf{SE.D})$ to be secure even against adversaries who learn up to $\kappa$ bits of the secret key. We here use this property to let the reduction ask the IND-CPA challenger for the bits of $\sigma^B[j]$ for all $j : (j,b) \in \mathcal{L}$.

This concludes the description of our simulation strategy. It can be seen that by construction we now have a simulator that does not use the input of the honest party B and we have argued for indistinguishability after each individual change. The complete description of the simulator after all the hybrids can be found in Figure 6. The simulator is simply a compilation of all the individual changes done in the above hybrids. Therefore the distribution of hybrid 11 is identical to the distribution of the simulation.

*Hybrid 12.* After encrypting 0s instead of the real input of B, it can easily be seen that there is only one place left where we use the input of B, namely to compute the set $I^B$ for which we need the input of B to evaluate the garbled circuits, as a bad circuit might give an output on some of Bs inputs and $\perp$ on some other inputs. We get rid of this last use of the inputs of B by removing the restriction that $|I^B| \leq \psi\kappa$ in the functionalities for *filling-in* and dropping the abort condition in Step 11 in Stage 3. This change is indistinguishable, as $|I^B| \leq \psi\kappa$ except with negligible probability. To see why this is the case, let $\mathsf{good}$ be the set of honestly generated circuits among those received by $B$. Thanks to the *input verifiability* property of our garbling scheme we know that for all $i \in \mathsf{good}$ the values received by B as output from the Stage 3 satisfy $y_i^B \neq \perp$.

Since we open $\epsilon\ell$ of the circuits in the cut-and-choose the probability that $\psi\kappa$ bad circuits will all survive without any being detected is less than $(1+\epsilon)^{-\psi\kappa}$, and we have set $\psi$ such that $(1+\epsilon)^{-\psi\kappa} = 2^{-\kappa}$.

It can be seen that in Hybrid 12 we no longer use the inputs of B. This is therefore a legal simulation strategy. The simulator is given in Figure 6. It is just a compilation of the individual changes in the 12 hybrids, so by definition Hybrid 12 is identical to the distribution of the UC simulation. What remains is therefore only to argue that Hybrid 0 is indistinguishable from the real protocol. In Hybrid $i$ define an event $E^i$ as follows. Let $Y = \perp$ if the ideal functionality aborts and let $Y = \{y_i\}_{i\in[\ell]}$ be the outputs of the ideal functionality otherwise. Let $Y' = \perp$ if the protocol aborts and let $Y' = \{y'_i\}_{i\in[\ell]}$ otherwise. Let $E^i$ be the event that $Y \neq Y'$. To argue that Hydrid 0 is indistinguishable from the real protocol it is clearly enough to argue that $\Pr[E^0]$ is negligible. We have that $\Pr[E^{12}] = 0$ by construction: it has already been argued that since Hybrid 5 the outputs of the *filling-in* for the honest party are correct; and it has been argued that since Hybrid 8 the corrupt party can only input the correct outputs to the *forge-and-lose* functionality, which implies that either the outputs that B received before this subprotocol are the correct ones or B will receive $\sigma^A$ as a result of *forge-and-lose* and compute the right outputs in the clear.

It then follows from Hybrid 0 and Hybrid 12 being indistinguishable that $\Pr[E^0]$ is indistinguishable from 0, i.e., negligible.

## 4 Dealing With Long Inputs and Outputs

The previous protocol allows to compute $f : \{0,1\}^{n_I} \to \{0,1\}^{n_O}$ where $n_I$ is the input size and $n_O$ is the output size. In the previous sections we have, for simplicity, assumed that $n_I = n_O = O(\kappa)$. However in general $n_I$ and $n_O$ can be of size linear in $|f|$. This presents an issue in the forge-and-lose step, since the size of the circuit implementing the functionality is of size:

$$(\ell n_O + \ell n_I + |f|)\kappa = O(\ell|f|\kappa)$$

instead of $O((\ell + |f|)\kappa)$ as desired. We describe here two optimizations which allow to deal with this:

**Dealing with Long Outputs:** It is quite easy to deal with long outputs in the following way: modify the circuit to be garbled so that, in addition to outputting $y_i$, it also outputs $h(y_i)$ with $h$ a collision resistant hash function. Now it is clear that $|h(y_i)| = \kappa < n_O$ and that now instead of letting A,B input the values $y_i$ to the forge and lose functionality, they input the hashes instead. The forge and loose finds the first index where the *hashes* differ, recompute the function (and the hash) on that index and determines who cheated.

**Deadling with Long Inputs:** The key here is to notice that only a single pair of ciphertexts is ever used during the forge and lose functionality, and the

only reasons for the parties to input all of the ciphertexts is to guarnatee that the parties do not learn in which positions (if any) the function is being recomputed. In some sense we are using a very naive private information retrieval (PIR), which can of course be replaced with a more clever one. So now we modify the forge and lose functionality in the following way: instead of having A,B input all ciphertexts at the beginning, they only input the outputs (or their hashes as described above). The funcitonality finds the first $i$ such that $y_i^A \neq y_i^B$ (if any), and then runs a 2-server PIR protocol with A,B. This allows the functionality to learn the ciphertext pair $X_i^A, X_i^B$ necessary to determine the right value of $y_i$ by receiving only $\sqrt{\ell} n_I$ bits from A,B. Since $n_I < |f|$ it is enough to assume that $\ell > |f|^2$ to bound this term with $\ell$.

All in all, the number of bits which A,B send to the forge and lose functionality is bounded by

$$(\ell\kappa + \sqrt{\ell} f + |f|)\kappa = O((\ell + |f|))\,\mathrm{poly}(\kappa)$$

as desired.

## References


[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 784–796, 2012.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513, 1990.

[Bra13] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, pages 441–463, 2013.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 537–556, 2013.

[FJNT15] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. Tinylego: An interactive garbling scheme for maliciously secure two-party computation. *IACR Cryptology ePrint Archive*, 2015:309, 2015.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 191–219, 2015.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 18–35, 2013.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 458–475, 2014.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 955–966, 2013.

[JS07] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *Advances in Cryptology - EUROCRYPT*

*2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, pages 97–114, 2007.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 1–17, 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, pages 52–78, 2007.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 476–494, 2014.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 579–590, 2015.

[MF06] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pages 458–473, 2006.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302, 2004.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, pages 368–386, 2009.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 223–238, 1999.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 220–250, 2015.

## A List of symbols:

- $\sigma^A, \sigma^B$ global encryptions keys, also used as selection bits used in $\mathsf{OTChoose}$;
- $\tau^A, \tau^B$ global MAC key, generated during *filling in*;
- $x_i^A, x_i^B$ the inputs used in execution $i$;
- $y_i^A, y_i^B$ the output received by A/B in execution $i$;
- $y_i$, the real output of execution $i$ (i.e., $f(x_i^A, x_i^B)$);
- We use $\hat{x}$ to indicate "garbled values/functions" (in the GC context);
- We use capitals for encryptions of the inputs under $\sigma$ $(X_A^i, X_B^i)$;
- $t_i^A, t_i^B$ are MACs computed during *filling in*;
- $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ the original function that we are trying to evaluate, with input/output size $n$;
- $\kappa$ the security parameter;
- $\ell$ the number of copies of $g$ we are evaluating i.e., $i = 1, \ldots, \ell$;
- $\epsilon$, the fraction of circuits being checked i.e., the number of garbled circuits which each party generates is $(1+\epsilon)\ell$, and then $\epsilon\ell$ of those are checked during the cut-and-choose. For simplicity we assume $\epsilon\ell$ to be an integer;
- $\psi = \log_{1+\epsilon}(2)$;
- Given a value $x$ $\langle x \rangle$ is a commitment to $x$ using randomness $\mathsf{open}(x)$.