Efficient Maliciously Secure Two Party Computation for Mixed Programs

Arash Afshar* Payman Mohassel† Mike Rosulek‡

January 26, 2017

Abstract

We propose a new approach for practical secure two-party computation (2PC) achieving security in the presence of malicious adversaries. Given a program to compute, the idea is to identify subcomputations that depend on only one or neither of the parties' private inputs. Such computations can be secured at significantly lower cost, using different protocol paradigms for each case. We then show how to securely connect these subprotocols together, and with standard 2PC yielding our new approach for 2PC for mixed programs. Our empirical evaluations confirm that the mixed-2PC approach outperforms state-of-the-art monolithic 2PC protocols for most computations.

1 Introduction

Secure two-party computation (2PC) allows two parties to compute a function of their private inputs, while revealing only the output of the computation and nothing more. A long line of work, initiated by Yao [Yao82], has culminated in increasingly practical techniques for 2PC based on *garbled circuits.*

In this work we focus on further improving the efficiency of garbled-circuit-based 2PC in the presence of *malicious* adversaries. Here the main technical challenge is to ensure that the garbled circuits used in the protocol are generated correctly. The standard way to ensure this is to use *cut-and-choose* [MF06, LP07, LP11, sS11, Lin13], in which the sender produces several garbled circuits and the receiver requests a random subset of them to be checked for correctness.

Thus the main (but not the only) source of overhead in achieving malicious-security is the *number of garbled circuits* generated. The last few years have seen significant progress in this metric. To achieve security level 2^{-s}, the state of the art has gone from $17s$ circuits [LP07] to roughly $3s$ circuits [sS11] to s circuits [Lin13]. We also point out that when repeating the same computation N times, the (amortized) number of circuits per computation can be reduced further to roughly $2+\mathcal{O}(s/\log N)$ circuits by performing a single cut-and-choose for all N evaluations' worth of circuits [HKK+14, LR14].

However, in some settings even *one garbled circuit* suffices for security against malicious adversaries! In particular, the protocol of Jawurek et al. [JKO13] achieves exactly this, for the special case where *the computation involves secret inputs from only one party.* For example, a zero-knowledge proof is a computation in which only the prover's input (the witness) is secret. Additionally, the protocol of [JKO13] requires garbled circuits with a weaker security definition, concretely costing half as much as standard garbled circuits [FNO15, ZRE15].

*University of Calgary. aafshar@ucalgary.ca
†Visa Research pmohasse@visa.com
‡Oregon State University. rosulekm@eecs.oregonstate.edu

Connector	Garbling Role	Connector Category	Complexity
A/B-Sec $\rightarrow$ 2PC	Same	Blind output & OT-based consistent input	$\mathcal{O}(m)$
	Different	Blind output & UHash-based consistent input	$\mathcal{O}(m + 2\mathsf{s} + \log \mathsf{s})$
A/B-Sec $\rightarrow$ A/B-Sec	Same	One-to-one reuse garbled keys	0
	Different	Plain connection	0
2PC $\rightarrow$ 2PC	Same	One-to-one reuse garbled keys	0
	Different	Double blind output & UHash-based consistent input	$\mathcal{O}(m + 2\mathsf{s} + \log \mathsf{s})$
2PC $\rightarrow$ A/B-Sec	Same	Many-to-one reuse garbled keys	0
	Different	Blind output & UHash-based consistent input	$\mathcal{O}(m + 2\mathsf{s} + \log \mathsf{s})$

Table 1: m is the output size of intermediate computation and s is a statistical security parameter.

1.1 Our Results

Motivated by the above, we classify computations in terms of the privacy of their inputs. A **plain** computation takes only public input; an **A-Sec** (resp., B-Sec) computation takes private input from *only* Alice (resp., Bob); a **2PC** computation takes private input from both parties.

In isolation, a plain computation can be done without cryptography; an A-Sec or B-Sec computation can be done securely using only one garbled circuit; a 2PC computation can be done using s garbled circuits. Note that for the purpose of this work we focus on the case of one-off computations, and do not consider efficiency gains by amortizing garbled circuits across many secure evaluations of the same computation. We emphasize that our approach provides significant improvement even in the case of single secure computations.

Since different classes of computations can be secured at vastly different costs, we consider the approach of decomposing a large computation in terms of many subcomputations, each of which is then characterized as plain, A/B-Sec, 2PC as above. Indeed, such a decomposition can be obtained from the SCVM compiler of [LHS+14] which compiles a program into a trace-oblivious version and identifies the sub-computations that do not depend on inputs of both parties.

With such a decomposition in hand, we then show how to securely "connect" different paradigms for malicious-secure 2PC, so that the cryptographic overhead is minimized at the fine-grained level of subcomputations. Note that for an A/B-Sec subcomputation the choice of the garbler is fixed in the protocol of [JKO13], while in a 2PC subcomputation (for which we use protocol of [LR15]) either party may be designed as garbler. Hence we describe connections among plain, A-Sec, B-Sec, and two variants of 2PC computations (based on which party is designated as garbler). Table 1 summarizes the various connectors we design and the additional cost they incur.

We also provide a full description of our construction for the example mixed 2PC of Fig. 12, where we use the construction of [JKO13] and the 2PC of [LR15] as building blocks, connected with the above-mentioned connectors, and prove its security.

We also implement our protocol in Java and use the SCAPI library [EFLL12] as our beck-end. Our prototype implementation can be configured to accept a program that is broken into smaller subprograms (i.e. mixed computation) or to accept a monolithic computation. Our experiments compare these two configurations for different program sizes and various ratios of A-sec/B-sec to 2PC computation. Our experiments confirm that our mixed 2PC protocol outperforms monolithic 2PC for ratios as small as 1.5, and can be a factor 17 or more faster when the ratio is over 60.

Technical challenges. Even if individual subcomputations are secure against malicious adversaries, their composition is not necessarily secure. For example, when computing $g(f(x))$, where f and g are distinct subcomputations, we must ensure that the data output by the f-computation is indeed what is used as input to the g-computation. Hence, the main challenge is to ensure the *integrity* of intermediate data in the "connective tissue" between each malicious-secure subcomputation.

A trivial way to ensure integrity is to add a MAC to all intermediate data. While conceptually simple, this solution would require adding MAC circuitry to every secure computation, which is a non-trivial overhead. To achieve minimal overhead requires more care, especially since, in general, the intermediate data is private to one or more of the parties.

Suppose that the output of subcomputation f is used as input to subcomputation g. In reality, f may have many outputs, each used in many places, and g may have many other inputs of various flavors. For simplicity here, we just consider a single output of f and a single input of g. We summarize two basic technical approaches taken in our main construction for connecting the output of f to input of g (different connectors and their costs are presented in Table 1):

- "Blinded output, consistent input": First, augment f to output $f(x) \oplus r_A \oplus r_B$, where r_A and r_B are random one-time pads provided by Alice and Bob, respectively. This blinded output $z = f(x) \oplus r_A \oplus r_B$ can then be made public, and the g-computation can be hard-coded with z and augmented to compute $g(z \oplus r_A \oplus r_B)$.

  Now it suffices to ensure that parties use the same r_A and r_B in both computations. In some connection scenarios we can use the same OTs to deliver garbled inputs to a party (i.e., in a single OT Alice receives garbled inputs for the first bit of r_A in *both* circuits). When this is not possible (e.g. when the garbling roles change), we use the universal-hash-based input consistency technique of [sS13], since unlike some other techniques, theirs can be used to ensure consistency of inputs in circuits *garbled by different parties.* The idea is to augment each circuit to compute $H(r)$, where H is a randomized hash function that both hides r but is chosen in a way that collisions are unlikely. After augmenting the f and g computations to compute appropriate hashes of r_A and r_B, the parties simply check that the (public) hashes agree.

  [sS13] show that this approach adds an overhead of $2s + \log(s)$ additional OTs, where s is a statistical security parameter and essentially no overhead within the garbled circuit when using the Free-XOR optimization of [KS08].

- "Reusing garbled values": When the same party is garbler for both the f and g computations, sometimes we can ensure integrity in an even simpler way. The idea is that at the end of the computation of f the Evaluator holds a *garbled representation* of $f(x)$. This representation has the property that is hard to guess the representation of any value other than $f(x)$ (i.e., it is hard to guess the complementary wire label on the output wires). Hence, the garbled circuits for g can use the same garbled representation / wire labels for its input wires.

  This approach of reusing wire labels has been used elsewhere [MGBF14, AHMR15], but not to connect different paradigms of protocols to the best of our knowledge. Using this kind of connection between subcomputations, the overhead is essentially zero.

There are significant technical challenges to overcome, apart from the high-level ideas above. We point out some of these challenges next.

Order of Execution. It is crucial to carefully coordinate certain events in different subprotocols. For example, in our first kind of connector, we must first perform the relevant OTs for inputs r_A, r_B in *both* the f and g computations. Only after r_A and r_B are committed in this way can the H hash functions be chosen.

Only then can f be evaluated to reveal z. Only then can g be garbled with z hard-coded. Similarly, in our second connector, when we use the same garbled labels for the output of f, and the input of g, we also have to use the same cut-and-choose for them. This means that the garbled circuits for both computations have to be sent before the choice of cut-and-choose is revealed, and the evaluation/opening for either subcomputation takes place.

Choosing the right order for garbling, opening, and evaluation become even more challenging when one considers the more complex graph of computation where each node is a subcomputation. In particular, we show that the required orders of execution can be in conflict with each other. We identify all the arising conflicts, and show a general way of removing them from any computation graph.

Use of JKO. The use of the protocol of [JKO13] in our construction also requires additional care. There is a phase in their protocol where the Evaluator holds a garbled representation of the output. But the garbled circuit is later *opened*, which destroys the authenticity property of this encoding (i.e., after this point it becomes trivial to obtain the garbled encoding of *any* value). This fact makes it unsuitable to reuse these wire labels, though it is possible to reuse wire labels for *inputs* to the [JKO13] protocol. Also, for technical reasons, we must use a slightly different approach than [JKO13] to prevent selective-failure attacks.

Cheating Recovery. Using a cheating recovery-based protocol such as [LR15] as the back-end for 2PC protocols is a great improvement in terms of the overhead of garbling for 2PC computations, but at the same time special care is needed to ensure that the cheating recovery phase is invoked properly when multiple computations are connected together. To reduce the overhead of the cheating recovery computation, we perform a single cheating recovery for a group of computations that share the same garbler and we handle cases where the cheating is detected in a subset of such computations. For example, consider the case where the evaluator needs to commit to the output of an A-Sec computation, but a cheating is detected at the previous 2PC computation. Now the question is how to make sure that the evaluator is still able to correctly commit to the garbled output of the A-Sec computation given the fact that the cheating recovery is performed after the A-Sec is evaluated.

2 Applications of Mixed Programs

Traditionally, computing a function or a program using 2PC involves converting the computation to a big monolithic logical circuit representation. As a result, computing big and complex circuits would not be possible as we would eventually run out of memory to hold all the garbled circuits required for this computation. To address this problem, one could use the pipelining technique of [HEKM11] or break the 2PC computation into a series of smaller 2PC computations and pass the state of the computation from one computation to the other [MGBF14]. Although both of these techniques fix the issue of having a large garbled circuit in memory, they both perform some unnecessary computations since in most of the real world scenarios, only parts of the program needs to remain private (i.e. computed using 2PC) and the rest could be computed plainly. Moreover, some other parts of the computation may only depend on the private data of one party and hence we can use only one garbled circuit to securely compute it [JKO13] as opposed to s garble circuits [Lin13].

We believe that as we move forward to apply secure computation to bigger functions or programs, the chance of finding such parts increases. This idea has also been explored in concurrent & independent work in [KMW16] and [Bau16], but with a different motivation; the motivation behind these works is input validity. Given a function $f(g(x), h(y))$ where x is the input of the first party and y is the input of the second party, the functions g and h can be seen as input validation functions that only depend on the input of one party and therefore can be computed securely with less garbling overhead. Both these work are restricted

to this special case while our work offers a general solution where party specific computations can appear at any stage of computation. An example scenario that is not covered in these works is performing a 2PC computation and returning party specific outputs, letting each party perform some computation on his own output, and finally performing another 2PC on the result of the newly updated output.

To break a program into smaller parts that depend only on public data or private data of only one of the parties, compilers such as ObliVM [LWN+15] or SCVM [LHS+14] can be used. In the following example we have used the SCVM since the current version of ObliVM can only identify public and non-public parts while SCVM can additionally identify parts that depend only on the private data of a party.

Examples. consider two products, p_1 and p_2 with high correlation such that customers who buy p_1, also buy p_2 (and vice-versa) with a high probability (e.g. wine and cheese, cellphone and cellphone cover/protector, etc). Assume these items are produced by $prod_1$ and $prod_2$ respectively, each with their own set of wholesale customers. From a business perspective, the two companies are interested to collaborate and attract the customers of the other company (e.g. offer a deal for customers that buy both products together). It is fair to assume that they are interested in such a collaboration only if they are able to attract new, profitable customers. Lets define "new" as a customer that is not in the intersection of the customer list and "profitable" as a customer that spends a certain amount buying products. From a security perspective, the two companies want to keep the list of their customers private (except for the ones that are common between the two) and only reveal the name of a single customer who is profitable to the other company if the other company can offer a profitable customer in exchange. Moreover, the companies want to ensure that the other party is telling the truth about their customers and their worth.

We implemented a program that performs the above in SCVM and as expected, the result (with slight modifications) is as follows (depicted in Fig. 1).

- Verification phase: Each party performs an A-Sec or B-Sec computation, proving that their input satisfies a property against a public value (e.g. an example of a relevant public value could be the company's public annual income).
- Set intersection phase: The parties run a 2PC computation that produces three outputs; 1) a public part, consisting of the name of the customers shared between the two. 2) a private part for $prod_1$, containing the list of customers of $prod_1$ that are "new" for $prod_2$. 3) a private part for $prod_2$, containing the list of customers of $prod_2$ that are "new" for $prod_1$.
- Identify most profitable: Each party runs an A-Sec or B-Sec computation to identify a "new" customer for the other party (e.g. finding the maximum paying customer).
- Checking customers worth: Finally, the parties run two small 2PCs, each checking whether the new customer offered by $prod_b$ meets the minimum requirements of $prod_{1-b}$.
- Reveal the customers: Finally, the parties run a 2PC that would reveal the new customer names to both parties if both conditions are satisfied.

A similar pattern can be seen in other applications too. For example, consider two moving points in a 2D plane. Where the position of the points, the speed, and direction of the movement is private and the goal is to compute their distance at a publicly known timestamp. In this computation, each party computes the new position of their point using an A-Sec or a B-Sec computation and they will run a 2PC to compute the distance of the new positions. Now, we can imagine a scenario where the 2PC could also give each party instructions on how to adjust their speed. In that case, the computation for adjusting the speed could be run as an A-Sec or B-Sec .

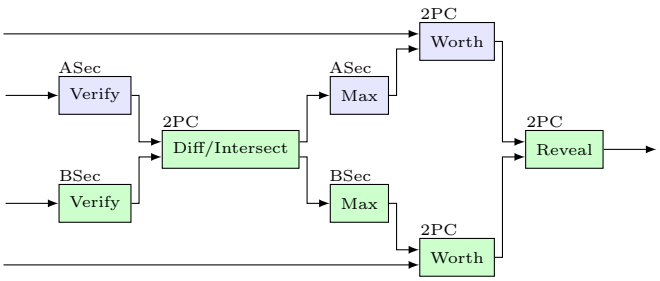

Figure 1: Example of a Mixed Computation, where different colors show different garblers

The example of Fig. 1 serves as a motivating example, but to demonstrate the proof of the full protocol, we have opted to use a simplified example that contains a sample of all of our connectors. We have also used this simplified example as a basis for our prototype implementation. The details of this example are described in section 6.

3 Preliminaries and Notations

security label. To specify which party can view a value, we define four distinct security labels: {Alice, Bob, Both, None} meaning that the data is visible only to Alice, only to Bob, both of them (i.e. the data is plain or public), or none of them (e.g. the data is the intermediate result of the computation).

We use the terms program, function, and computation interchangeably when the context is clear and assume that a program is already broken into smaller programs (or computations). We classify these computations by the security label of their inputs into 4 different computation types:

1. Plain: In this computation, the inputs and outputs are public values known to both parties (i.e. security label Both). Since input is visible to both parties, the output must also have the security label Both.

2. 2PC: In this computation, the inputs of both parties are private while the output can have any security label. Looking ahead, this computation is implemented as a maliciously secure two party computation using the cut-and-choose paradigm.

3. A-Sec: In this computation the input labels are Alice or Both meaning that they depend on first party's (Alice) private data and/or public data. The output of this computation is either private to Alice or is a public value, known to both parties (security label Alice or Both). Looking ahead, this computation is implemented using a modified version of the garbled circuit-based zero-knowledge proof of [JKO13]. In [JKO13] the prover has a private input w and the goal is to prove $y = f(w)$, where both y and f are public values and at the end of the computation, the verifier learns a Boolean value showing whether y is equal to $f(w)$. In our setting, both w and y are private inputs of the prover. I.e. the prover wants to show that he has correctly performed the computation f on its private input. Moreover, y might be passed to another computation. Therefore, we make the following two modifications to the [JKO13] work.

   (a) y is considered the private input of the prover.

   (b) the computation also outputs y.

4. B-Sec: This computation is very similar to A-Sec, with the difference that the private data belongs to the second party, Bob.

Mixed Computations. A program (or a computation) is defined as a series of smaller computations which are evaluated in a certain order. We define a sub-computation as a function and represent it as a 4-tuple $(\mathsf{N}, \mathsf{In}, \mathsf{Out}, \mathsf{body})$ where N is the name of the function of this computation, In and Out are the sets of the input/output variables of this computation, and body describes the function of this computation. The sets In and Out are subsets of the global set of variable name, GVN. We use the dot (.) accessor to access the elements of a tuple. For example, given a function f, the name of the function is shown as $f.\mathsf{N}$ and $f.\mathsf{In}$ is used to access the set of its input variables.

The relation between two functions f_i and f_j is defined using the global variable name set, GVN, such that if the same variable appears in the output set of a function f_i and it also appears in the input set of another function f_j we say that function f_i is connected to function f_j. Our functions can have unrestricted fan-out (i.e. the output of a function can be connected to one or more other functions), but they must have a fan-in of one (i.e. for any two functions f_i and f_j, $f_i.\mathsf{Out} \cap f_j.\mathsf{Out} = \emptyset$.

For example, the program of Fig. 1 would appear formally as depicted in Fig. 2.

$$
\begin{aligned}
\mathsf{GVN} = \{&\mathsf{pub}_1 : \mathsf{Both}, \mathsf{set}_1 : \mathsf{Alice}, \mathsf{tresh}_1 : \mathsf{Alice}, \mathsf{verRes}_1 : \mathsf{Alice}, \mathsf{disJoint}_1 : \mathsf{Alice},\\
&\mathsf{max}_1 : \mathsf{Alice}, \mathsf{resWorth}_1 : \mathsf{None}, \mathsf{pub}_2 : \mathsf{Both}, \mathsf{set}_2 : \mathsf{Bob}, \mathsf{tresh}_2 : \mathsf{Bob},\\
&\mathsf{verRes}_2 : \mathsf{Bob}, \mathsf{disJoint}_2 : \mathsf{Bob}, \mathsf{max}_2 : \mathsf{Bob}, \mathsf{resWorth}_2 : \mathsf{None},\\
&\mathsf{finalOut} : \mathsf{Both}\}\\
f_1 = (&\mathsf{verify}, \{\mathsf{set}_1, \mathsf{pub}_1\}, \{\mathsf{resVer1}\}, \mathsf{body}_1)\\
f_2 = (&\mathsf{verify}, \{\mathsf{set}_2, \mathsf{pub}_2\}, \{\mathsf{resVer2}\}, \mathsf{body}_2)\\
f_3 = (&\mathsf{diff\_intersect}, \{\mathsf{resVer}_1, \mathsf{resVer}_2\}, \{\mathsf{max}_1, \mathsf{max}_2\}, \mathsf{body}_3)\\
f_4 = (&\mathsf{max}, \{\mathsf{disJoint}_1\}, \{\mathsf{max}_1\}, \mathsf{body}_4)\\
f_5 = (&\mathsf{max}, \{\mathsf{disJoint}_2\}, \{\mathsf{max}_2\}, \mathsf{body}_5)\\
f_6 = (&\mathsf{worth}, \{\mathsf{max}_1, \mathsf{tresh}_1\}, \{\mathsf{resWorth}_1\}, \mathsf{body}_6)\\
f_7 = (&\mathsf{worth}, \{\mathsf{max}_2, \mathsf{tresh}_2\}, \{\mathsf{resWorth}_2\}, \mathsf{body}_7)\\
f_7 = (&\mathsf{reveal}, \{\mathsf{resWorth}_1, \mathsf{resWorth}_2\}, \{\mathsf{finalOut}\}, \mathsf{body}_8)
\end{aligned}
$$

Figure 2: Formal representation of program in Fig. 1.

Note that based on the description of the different types of computation, the type of a computation can be inferred from the security label of its input variables. Therefore, we assume an implicit parameter type for each computation and use the same dot (.) accessor to refer to it. For example, we write $f.\mathsf{type}$ to get the type of the function f. In the case of the above example, $f_1.\mathsf{type}$ would return $\mathsf{A\text{-}Sec}$.

Due to the nature of our building blocks, some computations can be garbled by either of the two parties. Therefore, we define another implicit parameter $\mathsf{garbler}$ for each computation. In the example of Fig. 2, $f_1.\mathsf{garbler}$ will always return Bob. But depending on the configuration, $f_3.\mathsf{garbler}$ can return Alice or Bob. Note that we are reusing $\mathsf{Alice}, \mathsf{Bob}$ to prevent introducing extra notations.

With the above description, a program can be viewed as a pair $(\mathsf{funcList}, \mathsf{GVN})$ where $\mathsf{funcList}$ contains the sequence of subprograms and GVN is the global list of variable names which shows how the subprograms are connected to each other.

3.1 Security Definition

We prove security using the ideal/real simulation paradigm in presence of a malicious adversary. For a program $(\mathsf{funcList}, \mathsf{GVN})$ and input set of $\mathsf{In}_{\mathsf{alice}}$ and $\mathsf{In}_{\mathsf{bob}} \subset \mathsf{GVN}$, in an ideal world the program description and the inputs of the parties are sent to an ideal functionality $\mathcal{F}$. The ideal functionality would then perform the computation and send each party the corresponding outputs (including the outputs of subcomputations for which the parties are allowed to see the value). By definition, the view of each party in this world consists of their own input and the output received from $\mathcal{F}$. In the real world, there is no ideal functionality and a protocol π is implemented to securely realize $\mathcal{F}$. In protocol π the parties interact with each other and exchange messages and eventually obtain the outputs of the computation. Therefore, in the real world the

view of each party consists of all the messages that were sent and received by a party, including the input and output messages. We say the protocol π securely realizes the ideal functionality $\mathcal{F}$ for a given program, if the view of an adversary $\mathcal{A}$ in the real world is indistinguishable from the view of the adversary in the ideal world.

More formally, let $\text{REAL}_{\mathcal{A},\pi,s}(\mathsf{In_{alice}}, \mathsf{In_{bob}})$ show the view of the adversary $\mathcal{A}$ in the real world protocol π, where s is the statistical security parameter, and $\mathsf{In_{alice}}, \mathsf{In_{bob}}$ are the inputs of the two parties, Alice and Bob. Moreover, let $\text{IDEAL}_{\mathsf{S},\mathcal{F},s}(\mathsf{In_{alice}}, \mathsf{In_{bob}})$ show the "simulated" view of the adversary in the ideal world using a polynomial-time simulator S (i.e. the adversary in the ideal world). The security under this definition is defined in Theorem 3.1 (a modified version of the definition in [LP07]).

Theorem 3.1 *A protocol π securely realizes an ideal functionality $\mathcal{F}$ in presence of a polynomial-time, malicious adversary $\mathcal{A}$, if given a polynomial-time simulator S and a statistical security parameter s,*

$$\{\text{REAL}_{\mathcal{A},\pi,s}(\mathsf{x},\mathsf{y})\}_{\forall x,y} \overset{s}{\equiv} \{\text{IDEAL}_{\mathsf{S},\mathcal{F},s}(\mathsf{x},\mathsf{y})\}_{\forall x,y}$$

holds. Where the $\overset{s}{\equiv}$ means the distribution of the views of the adversary in the real and ideal world is computationally indistinguishable with the negligible probability of 2^{-s}.

3.2 Garbling Notations

We use the term garbled "keys" of a wire to refer to the two random values chosen to represent a wire's two possible truth values and the term garbled "value" of a wire to refer to one of the garbled keys that correspond to the actual plain value of the wire. The garbling notation we use is a modified version of the notation of [BHR12] (to specify the manually chosen garbled input and output keys) to represent a garbling scheme. We define a garbling scheme as a tuple $(\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ as follows. Given a function f and a computational security parameter k

- $F \leftarrow \mathsf{Gb}(1^k, f, E, D)$ creates a garbled circuit F using a set of garbled input keys E and garbled output keys D.
- $X \leftarrow \mathsf{En}(1^k, x, E)$ creates the garbled inputs values for an input x.
- $Y \leftarrow \mathsf{Ev}(F, X)$ evaluates the garbled circuit given the garbled inputs (i.e. the garbled values corresponding to the plain input values) and produces the garbled outputs.
- $y \leftarrow \mathsf{De}(Y, D)$ decodes the garbled values to their plain form.
- Correctness: $\mathsf{ev}(\mathsf{f},\mathsf{x}) = \mathsf{De}(\mathsf{Ev}(\mathsf{Gb}(\mathsf{f},\mathsf{E},\mathsf{D}),\mathsf{En}(\mathsf{x},\mathsf{E})),\mathsf{D})$.
- We require the garbling scheme to have privacy and authenticity properties [BHR12]. The privacy property states that given the garbled function F, the garbled input values X, and the decoding information (i.e. garbled output keys D), the Evaluator will not learn any more than the actual result of the computation $f(x)$. The authenticity property states that given the garbled function F and the garbled inputs X, a polynomial-time adversary cannot create a valid garbled output value $Y' \neq Y$, except with negligible probability.

3.3 Garbled Circuit Protocols

Since cut-and-choose 2PC and the GC-based zero-knowledge are essential to our work, we present a high level sequence of steps taken in these protocols. Later on, when we describe our protocol, we refer to these steps and describe how they need to be changed or re-arranged for our protocol in section 4 to work.

High level steps of cut-and-choose 2PC of [LR15]. In traditional cut-and-choose 2PC, the Garbler garbles $3s$ copies of the circuit and sends it to the Evaluator. The Evaluator chooses a subset of these garbled circuits to be opened and evaluates the rest. After evaluating the remaining garbled circuits, the Evaluator accepts the majority result as the output of the computation.

The recent approach of [Lin13] showed how to reduce the number of garbled circuits to s. The Evaluator randomly chooses whether to check or to evaluate each garbled circuit. At the end of the computation, if the Evaluator receives two conflicting outputs, he uses them as proof of cheating. The parties then run a smaller traditional 2PC in which the Evaluator enters the proof of cheating he has found and the Garbler enters his original inputs. If the proof of cheating is valid, this smaller 2PC reveals the inputs of the Garbler to the Evaluator, which enables him to perform the computation locally.

In [LR15], the authors have proposed an efficient protocol in amortized setting where a 2PC computation is run multiple times. They have used [Lin13] as the base of their protocol and have proposed more efficient techniques for checking the consistency of the inputs of the Garbler. In this work, we have modified the work of [LR15] for a single execution and we use it as the cut-and-choose protocol for 2PC. The full modified [LR15], hereon referred to as LR15, can be found in Fig. 15 in Appendix. To help present our protocols more easily, we have captured the high level steps of LR15 in Fig. 3.

$f_{Construct}$:	Decide on the circuit C computing computation f.
$f_{GarbIO\&OT}$:	Garble input and output wires and run OT for Evaluator to obtain his garbled input values.
f_{Cut}:	Commit to cut-and-choose choices.
$f_{Garble\&Commit}$:	Garble the circuits, compute the commitments and send them.
$f_{GarblerInput}$:	Check consistency of the Garbler's inputs.
$f_{EvalMain}$:	Receive the Garbler's garbled inputs and evaluate the Main computation.
$f_{EvalCheatRecov}$:	Evaluate and check the Cheating Recovery computation.
f_{Open}:	Check the opened circuits of the Main computation.
f_{Reveal}:	If necessary, reveal the plain output and the associated garbled values.

Figure 3: High level steps of 2PC of LR15 (see Fig. 15) for a function f.

Garbled Circuit-based Zero-Knowledge proofs [JKO13]. In the context of the JKO protocol, the verifier is responsible for garbling and sending a garbled circuit and the prover is responsible for evaluating it. The high level description of this protocol is presented is Fig. 4.

$f_{Construct}$	Decide on the circuit C computing f.
$f_{GarbIO\&OT}$:	The Garbler chooses the garbled keys for input wires and the Evaluator receives the garbled values corresponding to his inputs using OT.
$f_{Garble\&Commit}$:	The Garbler garbles and sends the circuit.
$f_{EvalMain}$:	The Evaluator evaluates and commits to the garbled output.
f_{Open}:	The Garbler opens the garbled circuit.
f_{Reveal}:	If the checks pass, the Evaluator opens the committed garbled outputs.

Figure 4: High level JKO protocol for a function f.

3.4 Other Notations

We use H to denote a universal hash functions. To randomize H, a random value r is chosen where the size of r is greater equal to $2s + \log(s)$ [sS13]. s is statistical security parameter defined in section 3.1.

4 Secure Connectors

Given a program that is broken into smaller subcomputations, we need a secure way of connecting the subcomputations. For some of the cases, this connection is relatively straightforward (e.g. when connecting a 2PC computation to another 2PC such that both computations have the same garbler). But for others, it is not straightforward (e.g. connecting an A-Sec computation in which Bob is the garbler to a 2PC computation in which Alice is the garbler). Moreover, as we will discuss in more details in section 5, looking at two computations in isolation is not enough. Therefore, after securely connecting two subcomputations, we should step back and take a look at the overall computations and make necessary adjustments.

In this section we introduce all connector protocols needed to connect the various types of computations to each other in a secure way. The purpose of these connectors is to securely connect the outputs of a computation to the inputs of the next computation while preserving the privacy and consistency of the intermediate values against a malicious party. Depending on the garbling role, the type of the current computation and the next one, a different connector is necessary. As mentioned in the introduction, in the context of garbled circuits, we consider two different approaches for designing connector protocols.

- The connectors that reuse the garbling keys. This approach is further broken into two connector protocol types.
    1. Where the connection is one-to-one (e.g. when a 2PC computation with s garbled circuits is connected to another 2PC computation).
    2. Where the connection is many-to-one (e.g. when a 2PC computation with s garbled circuits is connected to an A/B-Sec computation with one garbled circuit).
- The connectors that blind the output with a random value, reveal the blinded output, and check the consistency of the random blinding/unblinding values across different computations. This approach is also broken down into two connector protocol types.
    3. Where the consistency of the random blinding/unblinding values is enforced using universal hash function technique [sS13].
    4. Where the consistency is enforced through a single OT.

We start by describing the connector for each of the above 4 categories. We will examine the reasons behind the choice of each connector method and the complexity of each connector. This is meant to be a detailed explanation and will serve as the template for other connectors of that kind which will be described afterwards. In what follows, we used the modified single-execution protocol of [LR15] as the 2PC protocol, we believe that the techniques presented here can be easily extended to other cut-and-choose based protocols.

4.1 Re-using the garbling keys: One-to-one

Consider the case where a 2PC is connected to another 2PC and both computations are garbled by the same party. This is a common connector and has a straightforward solution: set the garbled inputs keys of the second computation to be the same as the garble output keys of the first computation [MGBF14, AHMR15]. As a result, after evaluating the first computation, the Evaluator will obtain a garbled value for each output wire which he can directly use in the second computation as garbled input. Due to the security of the garbling scheme, if the Evaluator knows one of the garbled keys corresponding to a wire, he cannot guess the other key (except with negligible probability) and therefore the Evaluator cannot change the output of the first computation. Moreover, due to the cut-and-choose, the garbler cannot switch the truth value of a garbled value between two computations.

The main changes to be made to our 2PC protocols are first to make sure that same garbled keys are used in the corresponding output wires of the first computations and the input wires of the second computation. Second, we need to ensure that the two computations use the same cut-and-choose choices so that the circuits chosen for opening in the first computation, align with their counterparts in the second computation. Third, we need to change the cheating recovery computation to include both 2PCs. For this, we remove the cheating recovery of the first computation and merge it with the cheating recovery of the second computation. Such that if the Garbler cheats in either of the computations, the Evaluator learns all the Garbler inputs to both computations. The details can be found in Fig. 5.

$[f_1, f_2]_{Construct}$:	The parties decide on C_1, C_2 with replication factor s computing f_1, f_2. The parties decide on C_2' with replication factor $3s$ computing the cheating recovery circuit for both C_1 and C_2.
$[f_1, f_2]_{GarbIO\&OT}$:	The parties choose the garbled keys for input and output wires of both computations, such that the output wires of C_1 that are connected to C_2 share the same garbled keys. Moreover, these garbled output keys do NOT include the truth value in them. The parties then run prob-resistant OT on direct inputs of both computations.
$[f_1, f_2]_{Cut}$:	The Evaluator's choice of cut-and-choose is used for both computations.
$[f_1, f_2]_{Garble\&Commit}$:	The Garbler garbles and sends C_1, C_2, C_2' along with necessary commitments.
$[f_1]_{GarblerInput}$:	The parties run F_{ExComAZK} on Garbler's input to C_1 and C_2' to prove that consistency of inputs to C_1 and the cheating recovery computation.
$[f_2]_{GarblerInput}$:	The parties run a different instance of F_{ExComAZK} of Garbler's input to C_2 but with the same set of evaluation circuit. This instance also includes the consistency checks for the cheating recovery circuit.
$[f_1]_{EvalMain}$:	Evaluate C_1 and obtain the garbled intermediate values and both the garbled and plain value for non-intermediate wires.
$[f_2]_{EvalMain}$:	Evaluate C_2 and obtain the output and/or proof of cheating.
$[f_1]_{EvalCheatRecov}$:	Removed: it is merged in the next cheating recovery computation.
$[f_2]_{EvalCheatRecov}$:	If cheating recovery is successful, the Evaluator will learn the direct inputs of the Garbler to both C_1 and C_2 and can compute the output of f_1 and f_2 locally.
$[f_1, f_2]_{Open}$:	Open and check C_1, C_2.
$[f_1, f_2]_{Reveal}$:	Reveal the non-intermediate output of C_1 and the output of C_2.

Figure 5: Connector protocol 1: reuse garbling keys in 2PC to 2PC (same garbler) on functions $z = f_1(x_1, y_1)$ and $f_2(z, x_2, y_2)$ where z is the intermediate value passed from f_1 to f_2.

Complexity. The cost of connecting the two computation is effectively zero.

4.2 Re-using the garbling keys: Many-to-One

This type of connector is used to connect an LR15-based protocol to a JKO-based protocol, where the garbler of both computations is the same. Therefore, the output of many garbled circuits should be connected to a single garbled circuit. To connect the two, for a given output wire j of the first computation, we set the garbled output keys of wire j to be the same across all garbled circuits of the first computation. We also set the corresponding garbled input key of the next computation to be the same garbled keys. Moreover, we need to ensure that the current computation is not opened until after the next computation is evaluated. Since opening the current computation would reveal both garbled keys of the output wires and as a result would enable the Evaluator of the current computation to pass a different result to the next computation.

This approach works since in this case, the intermediate value is either the private value of the Evaluator or it is a public value (since the next computation is either A-Sec or B-Sec) and hence revealing it to the Evaluator is safe. We can use the same garbled keys for the output wires and their corresponding input wire as long as we delay the opening of the first computation until after evaluation of the second computation. In other words, the 2PC opening phase should be delayed to at least until after the evaluation of the A/B-Sec computation. Moreover, since the garbler should not learn the intermediate output value, there is no point in

$[f_1, f_2]_{Construct}$:	The parties decide on C_1 with replication factor s computing f_1, C_1' with replication factor $3s$ computing the cheating recovery circuit for C_1, and C_2 with replication factor 1 computing f_2.
$[f_1]_{GarbIO\&OT}$:	The Garbler garbles C_1, C_1' with the following changes. For each output wire j in f_1, the Garbler uses the same garbled key in all s garbled circuits (the garbled output keys would still contain the truth value). Note: proof of cheating will be changed to match the one presented in [Lin13]. The parties run prob-resistant OT for inputs to C_1.
$[f_2]_{GarbIO\&OT}$:	The Garbler garbles C_2 and sets the garbled input keys of the intermediate wires to be the same as the garbled output keys of C_1. Parties run batch committing OT for the input to C_2 (i.e. x_2).
$[f_1]_{Cut}$:	Commit to cut-and-choose choices for C_1 and C_1'.
$[f_1, f_2]_{Garble\&Commit}$:	The parties garble C_1, C_1', C_2 and commit to input and outputs.
$[f_1]_{GarblerInput}$:	Prove consistency of the Garbler's input to C_1.
$[f_1]_{EvalMain}$:	Evaluate C_1 and either obtain the same garbled output value in all circuits (call this garbled output gz), or obtain two different but valid garbled output values in two different circuits call them gz_1 and gz_2, set $gz = gz_1$ (the choice is arbitrary). Proceed to C_2 using gz.
$[f_2]_{EvalMain}$:	Evaluate C_2 but do NOT commit to garbled output values yet.
$[f_1]_{EvalCheatRecov}$:	After the running the cheating recovery as per [Lin13], the Evaluator will obtain the Garbler's input to C_1 (if he can provide a valid proof of cheating) and can compute f_1 locally. *no-cheating*: the Evaluator accepts gz as the output of the C_1, and commits to the garbled output of C_2. *cheating*: The Evaluator locally computes f_1 and can decide which one of gz_1 or gz_2 is the correct garbled value for the result of f_1 and records that as gz. The Evaluator will proceed to evaluate C_2 once again using gz. He would then commit to the new result obtain from C_2.
$[f_1, f_2]_{Open}$:	the Evaluator opens and checks C_1 and C_2.
$[f_1, f_2]_{Reveal}$:	the Evaluator opens the commitment on the garbled output of C_2 but does not reveal the output of C_1.

Figure 6: Connector protocol 2: reuse garbling keys in 2PC to A/B-Sec (same garbler) on functions $z = f_1(x_1, y_1)$ and $f_2(z, x_1)$.

committing to the garbled output value.

We also need to adapt the cheating-recovery techniques to work with this connector. Note that in this case, the output is not plain, nor is it public. Nevertheless, it is visible to the Evaluator. Therefore, we can still apply the cheating recovery technique. After evaluating the first computation and obtaining the garbled output values, if cheating is detected, the Evaluator records the proof of cheating and continues the protocol (using dummy values) and evaluates the JKO-based computation. Then, the parties proceed to the cheating recovery phase through which the Evaluator learns the inputs of the garbler. Now, the problem is that the Evaluator should be able to commit to the garbled output of the JKO-based computation, for which he does not have valid garbled output values (since he performed the computation using dummy values). Note that given the plain inputs of the Garbler, the Evaluator can compute the output of each of the computations. Also note that the Evaluator has all the information necessary to evaluate the JKO-computation once again. So, the Evaluator starts from the first computation for which he detected a cheating. By definition, he must have two different but valid garbled output values. Knowing the plain output of the first computation, the Evaluator can choose the correct garbled output value and use that to evaluate the next computation. The reason this approach works is that the cheating recovery technique guarantees to have at least one correctly constructed circuit (except with negligible probability), and the question of identifying which of the garbled circuits are valid can be easily determined if one has the plain output for that circuit.

A detailed description of this connector is presented in Fig. 6.

Complexity. Similar to the previous case, the cost of connecting the two computations is zero.

$$
\begin{array}{lll}
f_1 = (C_1, \{x : \mathsf{A}\}, & \{y : \mathsf{A}\}, & \{y = C_1(x)\}) \\
f_1' = (C_1, \{x : \mathsf{A}, b : \mathsf{B}\}, & \{y : \mathsf{Both}\}, & \{y = C_1(x) \oplus b\}) \\
f_1'' = (C_1, \{x : \mathsf{A}, b : \mathsf{A}, r : \mathsf{A}\}, & \{y : \mathsf{Both}, h : \mathsf{B}\}, & \{y = C_1(x) \oplus b, h = H(b||r)\}) \\
 & & \\
f_2 = (C_2, \{y' : \mathsf{A}, q : \mathsf{B}\}, & \{z : \mathsf{Both}\}, & \{z = C_2(y', q)\} \\
f_2' = (C_2, \{y' : \mathsf{A}, q : \mathsf{B}, b : \mathsf{A}\}, & \{z : \mathsf{Both}\}, & \{z = C_2(y' \oplus b, q)\} \\
f_2'' = (C_2, \{y' : \mathsf{A}, q : \mathsf{B}, b' : \mathsf{A}, r' : \mathsf{A}\}, & \{z : \mathsf{Both}, h : \mathsf{B}\}, & \{z = C_2(y' \oplus b', q), h = H(b'||r')\})
\end{array}
$$

Figure 7: Blind output, consistent input. Alice label is shown as A and Bob label is shown as B due to space limitations

4.3 Blinded output, consistent input: OT-based

Consider the case of connecting an A-Sec to a 2PC where the garbler of both computations is Bob. An example of this case is depicted in Fig. 7. Without loss of generality, assume that C_1 is an A-Sec computation and has an input x owned by Alice and outputs an output y belonging to Alice . Also assume that this output is passed to a 2PC computation, C_2, along with some other input belonging to Bob. Computation C_2 will compute an output visible to both.

In this case, although the output belongs to Alice, we cannot reveal it to her and trust her to pass the correct value as input to the second computation. And, if we keep the output garbled, we cannot pass it to the next computation by reusing the garbled values. The reason is that the next computation is evaluated as a 2PC and therefore, if the input wires of the checked circuits have the same garbled keys as their counterparts in the evaluated circuits, the Evaluator will learn the inputs to the circuit.

Our solution for this problem is to first blind the output of the computation given a random value, b, chosen by the Evaluator. As a result, the output of the computation can be safely revealed to Bob (assuming everything up to this point is secure; more on this later). Bob can hard-code this public, blinded value in the next computation and augment the next computation to first receive the unblinding value from Alice and then proceed to evaluate the rest of the computation. This mechanism ensures the output of the computation is correctly encoded and decoded between the computations, assuming that the same random value is used for blinding and unblinding.

To enforce the consistency of the blinding and unblinding values, we use the same OT to transfer the garbled keys corresponding to blinding/unblinding value b in both computations. In Fig. 7, this approach is show in f_1' and f_2'. In the augmented computations f_1' and f_2', the blinding value and the unblinding value are shown with the same letter b in red to signify the fact that using OT, the garbled values delivered to Alice in both computations are the same. All that remains is to ensure that Bob is passing consistent garbled keys to the OT protocol (i.e. preventing selective failure attack). The mainstream techniques for preventing selecting failure attack is either to use a committing OT, or to use a probe-resistant OT [LP07]. In the case of this connector, the committing OT approach would be inefficient as it does not play well with OT Extension. In the JKO paper, the Garbler's input to all the OTs are opened, and hence the Garbler can simply commit to the randomness it uses in the OT extension and open the commitment at the end. But in our case, when using the same OTs for both JKO and an LR15-based 2PC, it is not clear how to selectively open a subset of the OTs, while using OT extension.

Therefore, to be able to make use of the efficiency gains of OT extension, we use the probe-resistant OT in this case. Definition 1 describes this method of preventing selective failure attack. The idea is to expand the users input (i.e. each bit is expanded to many bits whose XOR will result in the actual bit) such that if the malicious Garbler tries to cheat in any of these new expanded bits, he will be caught will probability 1/2 for each expanded bit. The definition 1 defines a s-probe-resistant matrix where s is the statistical security parameter and n is the length of the input to be expanded using this matrix. By a prob-resistant OT, we mean an OT protocol that first expands the inputs of the Evaluator by an s-prob-resistant matrix, runs the

OT protocol on the expanded inputs, and finally computes and returns the garbled values corresponding to the Evaluator's non-expanded input wires.

As for cheating recovery, it is performed without any change. The details of this connector are presented in Fig. 8.

Definition 1 ([LP07], [sS13]) *The matrix $M \in \{0,1\}$ is called s-probe-resistant if for any $L \subseteq \{1,2,\ldots,n\}$, the Hamming distance of $\oplus_{i \in L} M_i$ is at-least s, where M_i denotes the ith row vector of M_i.*

$[f_1, f_2]_{Construct}$:	The parties decide on $C_1(x_1, b_1)$ with replication factor 1 computing $z' = f_1(x_1) \oplus b_1$, they also decide on $C_2(z', b_2, x_2, y_2)$ with replication factor s computing $f_2(z' \oplus b_2, x_2, y_2)$ and C_2' with replication factor $3s$ computing the cheating recovery for C_2.
$[f_1, f_2]_{GarbIO\&OT}$:	The parties choose the garbled keys for the input and output wires based on the corresponding protocols with the exception of z'. The garbled input keys for z' are chosen such that the garbled keys include the truth value as their first bit. The parties run the appropriate OT for the inputs except for z', b_1, b_2. For z' we won't need an OT as the value will be provide in plain. The garbled values for b_1 and b_2 are passed in the same prob-resistant OT. (*Note:* For better efficiency, z' can be hard-coded in C_2. This removes the extra input commitments on z' wires but adds a restriction that C_2 can be garbled only "after" C_1 is evaluated and checked).
$[f_1]_{Garble\&commit}$:	Garble C_1.
$[f_1]_{EvalMain}$:	Evaluate C_1 and commit to the output, z'.
$[f_1]_{Open}$:	Open and check C_1. The garbler also sends the randomness necessary to open and verify the OT on b_1.
$[f_1]_{Reveal}$:	Open the commitments on output and reveal z'.
$[f_2]_{Cut}$:	Perform as instructed by the protocol.
$[f_2]_{Garble\&commit}$:	Garble C_2 and C_2'. (*Note:* if z' was not coded as input wire, then in this stage it would be hard-coded in C_2).
$[f_2]_{GarblerInput}$:	Perform as instructed by the protocol.
$[f_2]_{EvalMain}$:	Perform as instructed by the protocol.
$[f_2]_{EvalCheatRecov}$:	Perform as instructed by the protocol.
$[f_2]_{Open}$:	Perform as instructed by the protocol. Also, open and verify the OT for b_2 input wires.
$[f_2]_{Reveal}$:	Perform as instructed by the protocol.

Figure 8: Connector protocol 3: blind output, consistent input using OT for A/B-Sec to 2PC (same garbler) on functions $z = f_1(x_1)$ and $f_2(z, x_2, y_2)$.

Complexity. The circuit augmentation is only XOR gates which are almost free [KS08]. The extra costs are the cost of a prob-resistant OT on the blinding and unblinding values, which is $\mathcal{O}(m)$ OTs where m is the length of the output of the computation. This cost can be significantly reduced using OT extension [IKNP03].

4.4 Blinded output, consistent input: Universal hash-based

Consider the same scenario of connecting an A-Sec to a 2PC. This time, assume that the garbler of the 2PC is Alice. Therefore, in this case we have different garblers for each computation. We will use the same general technique of blinding the output and checking the consistency of the input with an important difference. Note that unlike the previous case, since the garbler of the two computations are different we cannot rely on OT to check the consistency of the input. Instead we use a universal hash function to compute the hash of the blinding value and the unblinding value and compare the results. In Fig. 7, the augmented computations f_1'' and f_2'' are passed a blinding value b and an unblinding value b' (in blue) to signify the fact that Alice may try to input different values in different computations and she will be caught (with high probability) if the result of the universal hash function is different.

The hashing function we use is the universal hashing of [sS13] which was originally used as an efficient tool for checking garbler's input consistency in a single 2PC computation. To have a collision-free universal

hash function with hiding properties, the hash function should be randomized and the inputs to this hash function should be fixed (i.e. committing to the inputs) before the hash function is sent to the garbler. Therefore, using it in our setting where the garbling role changes is a bit tricky. To better understand the requirements, lets revisit them: Alice (i.e. the party who owns the blinding and unblinding random inputs) should fix her inputs and only "after" that, Bob (i.e. the party who will check the result of the hash function) should choose the hash function. Note that Alice should fix her input in both computations since the hash function will be the same for the current computation and the next one. Fixing an input of a computation for which Alice is the garbler is straightforward: she simply commits to her garbled inputs as specified in [sS13]. But, if she is the Evaluator of a computation (e.g. for the first computation which is an A-Sec computation), fixing input can be achieved by running the OT protocol on the inputs.

Therefore, the order of computation for this kind of connector is as follows. Alice chooses the blinding and unblinding values and Bob chooses the universal hash function. Alice is the Garbler of the first computation and Bob is the Garbler of the second computation. Alice chooses the garbled keys for the wires corresponding to the blinding values for the first computation. She will then commit the garbled values corresponding to her choice of blinding values and sends the commitments to Bob. For the second computation, Bob chooses the garbling keys for the wires corresponding to the unblinding values and the parties run OT. At this point Alice is committed to her choice of blinding and unblinding values and it is safe for Bob to send the chosen hash function to Alice. The two parties will then augment the computation for which they are the garbler and garble them. In Fig. 7 this is shown as adding the hash function H to the body. Finally they proceed to evaluation and opening phase. The details of operations is shown in Fig. 9. Unlike the previous connector types, the garbling role is different in the two computations, therefore in Fig. 9, we have described the case where Alice is the Evaluator of the first computation and the garbler of the next computation.

As for cheating recovery, it is performed for the 2PC computation without change.

Complexity. To randomize the universal hash function, we need to choose a random value of length at least $2\mathsf{s} + \log \mathsf{s}$ for the security guarantee of 2^{-s}, appended to the original input of the hash function. Assuming an output size m for the first computation, the blinding and unblinding values would also have a size m. Therefore, the extra input size will be $m + 2\mathsf{s} + \log \mathsf{s}$. The hash function is composed of only XOR gates [sS13] and using Free-XOR [KS08], we garble and evaluate it almost for free. The extra costs are the commitments on the input of the hash function and the cost of OT for these extra inputs, which totals to $\mathcal{O}(m + 2\mathsf{s} + \log \mathsf{s})$ commitments and $\mathcal{O}(m + 2\mathsf{s} + \log \mathsf{s})$ OT queries. Using OT extension we can significantly reduce the cost of OT to only inexpensive symmetric operations.

4.5 Other connectors

The connector protocols for all other combination of computation can be easily reduced to one of the above-discussed connector types. Table 1 shows the connector type that is used for each possible scenario and the cost of each connection. For the most part, the choice of the connector and its costs follows the above description of connector categories. Therefore, in this section we will only talk about connectors that need extra reasoning.

2PC to A/B-Sec , different garblers. In this case, the current computation is a 2PC. Assume that the next computation is an A-Sec computation (Bob is the garbler and Alice is the Evaluator). In this case the Evaluator of the first computation, Bob, does not and should not know of the output of the computation as it may contain private information of the Alice as per A-Sec definition. For this case we first blind the output and then verify the consistency of the blinding using the universal hash function. The cheating recovery is performed normally.

	Note: Bob is the Garbler of f_1. Alice is the Garbler of f_2.
$[f_1, f_2]_{Construct}$:	The parties decide on $C_1(x_1, b_1)$ with replication factor 1 computing $z' = f_1(x_1) \oplus b_1$, they also decide on $C_2(z', b_2, x_2, y_2)$ with replication factor s computing $f_2(z' \oplus b_2, x_2, y_2)$ and C_2' with replication factor $3s$ computing the cheating recovery for C_2. Bob decides on a Universal Hash function H but does not send it to Alice. The input and output length of H is publicly known by both parties and the input and output wires corresponding to H in each of the computations are shown by $IN(f_i, \mathrm{H}), OUT(f_i, \mathrm{H}), i \in \{1, 2\}$.
$[f_1, f_2]_{GarbIO\&OT}$:	The party responsible for garbling of each computation garbles the input and output garbled keys as instructed by the corresponding protocol (e.g. inputs garbled keys for $IN(f_2, \mathrm{H})$ are chosen normally and output garbled keys for $OUT(f_2, \mathrm{H})$ are chosen to include the truth value). As in the previous connector, the public value z' can be handled by either including the truth value in the garbled key, or hard-coding it in C_2 after C_1 is evaluated and checked. *commit to unblinding values in* C_2: Alice commits to the garbled values for her choice of unblinding value in C_2 (for input wires $IN(f_2, \mathrm{H})$) using different seeded PRF than the one used to generate the garbled keys to derive the necessary randomness as per [sS13]. *Send the hash function* Alice is now committed to her choice of blinding and unblinding in both computations. Therefore, Bob sends her the universal hash function H he has chosen.
$[f_1]_{Garble\&Commit}$:	The Garbler of the first computation garbles C_1 and H.
$[f_1]_{EvalMain}$:	The Evaluator of the first computation evaluates C_1 and H and commits to their outputs.
$[f_1]_{Open}$:	Open and verify both C_1 and H.
$[f_1]_{Reveal}$:	Open the commitments on the output of C_1, i.e. the value z', and H.
$[f_2]_{Cut}$:	Performed as instructed by the protocol.
$[f_2]_{Garble\&Commit}$:	The Garbler of the second computation garbles C_2, C_2', H. If needed, he hard-codes z'.
$[f_2]_{GarblerInput}$:	Performed as instructed by the protocol. The inputs to H are also passed to $F_{\mathsf{ExCom}\Delta\mathsf{ZK}}$ although their consistency is checked using the hash function itself. The reason for passing them is to ensure that they are consistent with the inputs passed to the cheating recovery computation.
$[f_2]_{EvalMain}$:	Evaluate both C_2 and H.
$[f_2]_{EvalCheatRecov}$:	If the garbler cheated, Bob learns x_2, b_2 and can compute $f_2(z' \oplus b_2, x_2, y_2)$ locally.
$[f_2]_{Open}$:	Performed as instructed by the protocol. Moreover, Bob checks that the output of the hash function of the second computation that he just evaluated, is the same as the output of the hash function of the first computation.
$[f_2]_{Reveal}$:	Reveal the output of C_2 and the output of H.

Figure 9: Connector protocol 4: blind output, consistent input using universal hash function: A-Sec to 2PC, different garbler. For functions $z = f_1(x_1)$ and $f_2(z, x_2, y_2)$.

2PC to 2PC, different garblers. This case is similar to the case of connecting a 2PC to an A/B-Sec with the difference that the output of the first computation may contain private information about "both" parties. Therefore, a single blinding is not sufficient, rather both parties should blind the output and therefore the hash of the blinding of both parties should be computed and checked in both computations. After choosing their own random blinding values b_{Alice} and b_{Bob}, and committing to them in the computation that they are responsible for garbling, both parties send the committed values to the other party. They will then run two instances of OT protocol. Next, each party chooses a hash function and sends it to the other party. Finally, they both garble their augmented 2PC with both hash functions. Therefore, the cost is twice the cost of the connector type 4, described in section 4.4. The cheating recovery is performed normally. Alternatively, after committing to the inputs of the universal hash function, the parties can run a coin-tossing protocol and choose a single hash function to be used to check the consistency of the blinding values of both parties.

Evaluating Plain Computations Parties evaluate the public part on their own (Note that the case of A-Sec to B-Sec and its inverse also fall under this category. Since by definition, this connection is only allowable if the output that is passed between the two computations is a public value). After computing the public value, the Garbler of the next computation hard-codes this value in the next computation. The evaluating party checks the correctness of the hard-coded value in the checked circuits. An alternative solution is to

include the truth value in the garbled input keys. Therefore, after revealing the plain inputs to both parties, the garbler of the next computation can open the commitment on the correct garbled input value and the Evaluator can verify the truth value. In our implementation we have used this alternative approach since it results in a more straightforward implementation but at a small hit to performance.

4.6 Multiple Connectors

As described in section 3, a computation can have more than one type of output and each output can be used as input to several other computations. If two different outputs are mapped to two other computations, the computations are augmented based on the requirements of both connectors. If the same output is connected to more than one other computation, then the current computation is augmented to output two copies of the output, and each set of the output wires is connected to the corresponding next computation using the appropriate connectors.

5 Order of Evaluation and Checking

In order to securely compose the connector protocols we discussed earlier, we need to maintain the following invariants.

- Alice only learns the values of the output wires labeled Both or Alice (similarly for Bob).
- Given that various computations may perform opening/check step at different stages, each party should only learn the output of a computation when the "checking" for all previous computations that influence the current one are already passed.

For example, for connectors that blind output and check consistency of the blinding, before revealing the blinded value, we should open and verify the circuit. Therefore, the Evaluator should first commit to the Garbled output values that he has obtained and send them to the Garbler. Then, the garbler would send the opening of the circuit. If all the checks pass, evaluator would send the opening for the commitments to the Garbled output values. Finally, the Garbler will send the translation table to the Evaluator. In the case of the connectors that reuse the garbled value, the parties evaluate both computations, then open both computations (starting with the computation which appeared earlier in the funcList).

But when combining different connectors, the order of opening and evaluation is not straightforward and needs extra care. Consider the following two cases.

- Fig. 10 (a): The Garbler of the 2PC computations are the same. Therefore, as per connector type 1 described in section 4.1, all the computations should be garbled and sent, before the Evaluator can reveal the cut-and-choose choices. But note that, as per plain connector described in section 4.5, the computation that created the Plain value should be evaluated and verified, and if the checks pass the value is revealed. After that, the Plain value is hard-coded in the next computation.

  To summarize, we need to garble and send the computation C_3 along with C_1 and C_2, but we cannot do that unless C_1 is evaluated and opened. This creates an interdependency. To break it, we can change the garbling role of any of the 2PCs.

- Fig. 10 (b) has the same problem. Note that in contrast to the previous case, we cannot break the interdependency by changing the garbling role of C_1, since we would end up in a similar situation again. To break the interdependency, instead of reusing the garbled values for the case of 2PC to A-Sec, we use a connector that blinds the output. The most appropriate connector is the one that was used to connect A/B-Sec to 2PC, same garbler.

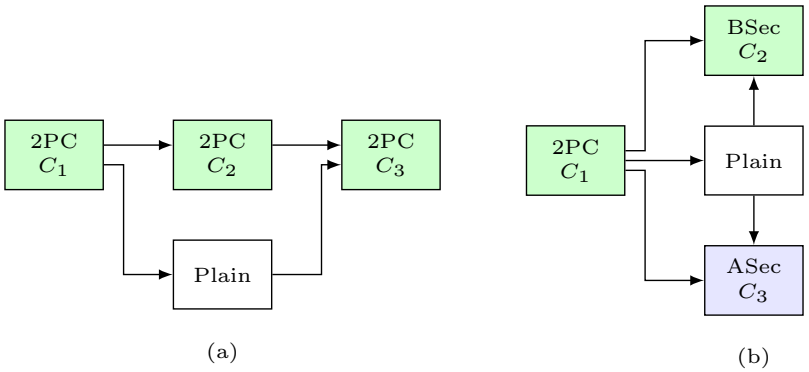

Figure 10: Interdependency in sub-computations.

We generalize the above examples by modeling the computations as nodes and connectors as directed edges of a graph. In this graph, edges have colors based on the type of the connectors. If the connector is reusing the garbled value and the garbler is Alice, the edge is colored green and if the garbler is Bob, the edge is colored blue. For other connectors where the output is made public, the edge is colored white. An interdependency can only happen when the output of a computations is split and then re-combined. I.e. given two nodes, C_i and C_j of the graph, if there are more than one path between the two nodes, there is a potential for an interdependency. If a path from node C_i to C_j consists of edges where the colors are all green or all blue, then all edges of all other paths from C_i to C_j should be of the same color. Otherwise, the two nodes are interdependent.

As shown in Fig. 11 (a.1), there are two paths between C_1 and C_3, one is all green and the other is all white. Therefore, there is an interdependency. By toggling the garbler of C_1 (Fig. 11 (a.2)), one path consists of all white edges, and the other consists of white and green edges which shows that the nodes are no longer interdependent. Similarly, Fig. 11 (b.1), a path from C_1 to C_2 is all green, but after changing the connector, all paths are white.

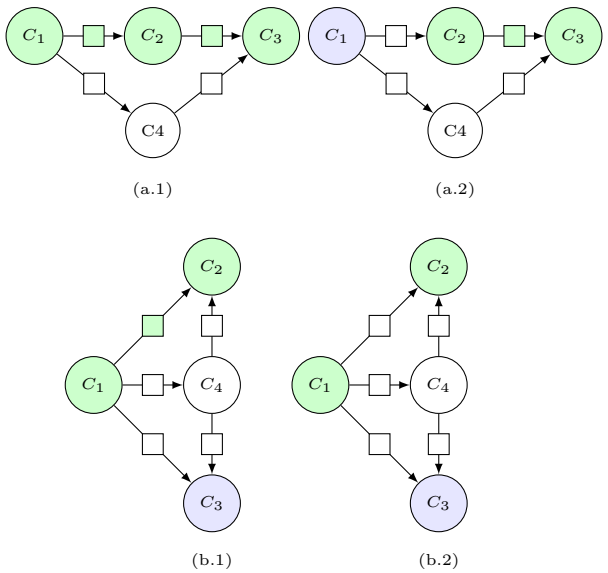

Figure 11: Breaking interdependency in computation graph.

To summarize, in order to remove the interdependencies, we apply one of the following transformations on the graph.

1. Change the garbling role of 2PC computation, or
2. Change the connector type of a connection.

After removing the interdependencies, we determine the order of evaluation and opening as follows. First, we cut the graph at white edges. The result will be a group of connected components. We define inputs of a component as the set of all white edges entering the component and the output of a component as the set of all white edges leaving a component.

Each connected component is essentially a group of computations that are connected using connectors that reuse garbled values. Therefore, all the computations of a connected component are garbled and evaluated first, then the Evaluator commits to all the outputs of the connected component. Afterwards, all the computations inside the connected component are opened are verified. If all the checks pass, the commitments on outputs of the component are revealed. Note that all the computations inside a connected component share the same cut-and-choose choices. The following sections describe this in details.

5.1 Inside a connected component

When two connectors appear after each other, they have a computation in common. The goal is to create a new connector that has the steps of both connectors and not to perform a task (like garbling) on their shared computation twice.

Inside a connected component we only use the 1st and 2nd connectors (Figures 5 and 6). There are two important changes that need to be made when merging the two connectors. First, moving the cheating recovery to the very end: after garbling and evaluating all the computations of this connected component. Second, if two or more 2PCs follow each other, they need to use the same choice of cut-and-choose.

Consider the following three different ways that connector can appear after each other inside a connected component.

1. 2PC $\rightarrow$ 2PC $\rightarrow$ 2PC (i.e. 1st connector followed by the 1st connector)
   Both connectors perform $Construct, GarbleIO\&OT, Cut, Garble\&Commit$, but perform a single "Cut" for all three computations. We also remove the cheating recovery related parts of the first two computations and modify the cheating recovery step of the last computation to reveal the inputs of all three computations. This means adding calls to $F_{\mathsf{ExCom}\Delta\mathsf{ZK}}$ between each computation and the cheating recovery computation to check the Garbler's input consistency.

2. 2PC $\rightarrow$ 2PC $\rightarrow$ A-Sec and similarly for B-Sec (i.e. 1st connector followed by the 2st connector)
   The cheating recovery computation of the first connector is removed and the cheating recovery computation of the second connector is changed to also reveal the Garbler's input to the first computation.

3. 2PC $\rightarrow$ A-Sec $\rightarrow$ A-Sec and similarly for B-Sec (i.e. 2nd connector followed by the 1st connector)
   Similar to the second case. The cheating recovery is moved to the very end.

5.2 Between connected components

In this case, the two connectors also share a computation. There are 4 different computations that we need to examine: $\alpha_1 \rightarrow \alpha_2 \rightarrow \beta_1 \rightarrow \beta_2$. Where $\alpha_1 \rightarrow \alpha_2$ are connected inside a connected component, $\beta_1 \rightarrow \beta_2$ are connected inside another connected component, and $\alpha_2 \rightarrow \beta_1$ is the bridge between the two connected components. The challenge here is to merge the steps of the connector for $\alpha_2 \rightarrow \beta_1$ with the other two

connectors such that we do not have to make drastic changes to the template we just defined for connectors inside a connected component.

To connect two connected components, we use the 3rd and 4th connectors (Figures 8 and 9). Note that in both of these connectors we first construct the computations, choose the garbling keys for input and output wires and perform OT. Afterwards, there is a clear separation between steps of the first computation and the second. Therefore, to merge the connectors we first modify the connectors for $\alpha_1 \rightarrow \alpha_2$ to include the necessary changes in the circuit structure of α_2 and β_1 (e.g. adding blinding, unblinding, and universal hash) and merge the $garbIO\&OT$. Note that to pass the plain output of α_2 to β_1, we have two choices: 1) choose the garbling values to include the truth value and 2) hard-code the value after evaluating and checking α_2. Merging the steps for the first case is trivial. The second case is also doable, but describing it is more complicated. Here, we assume the first method is used.

Note that this merge implies that constructing the circuit and running OT for α_2 should happen at the same time as β_1 (each in different connected component), which by extension means that for "all" the computations in our program, regardless of which connected component they belong to, we first construct the computation (i.e. convert it to Boolean circuit and perform any augmentation necessary as per the connectors), then we garble all the input and output wires, and run the OT. This turns out to be in our favor, since as will be described in the Appendix, to be able to prove the security of the protocol we need to extract all the inputs first, and then proceed to the computation so that the parties cannot adaptively choose their input after learning the intermediate outputs.

After $Construct$ and $GarbIO\&OT$, we merge the steps of α_2 and reveal the plain output. At this point we are done with everything related to the first connected component (except for the output of the universal hash function) and we move on to the second connected component and merge the steps of β_1. In the case of checking consistency using universal hash function, the plain output of α_2 is safely revealed but the output of β_2 cannot be revealed unless the hash values match. In other words, as part of the checking of the β_2 in the steps of the second connected component, we compare the output of the hash function with the one recorded from the previous connected component.

Regarding the cheating recovery, our goal is to have "one" cheating recovery computation per each connected component. Therefore, in the case where the 4th connector type is used to connect $(\alpha_2 = \mathsf{2PC}) \rightarrow (\beta_1 = \mathsf{2PC})$ different garblers, the cheating recovery computation of α_2 is NOT merged with the cheating recovery of β_1. Instead, their cheating recovery is merged with the cheating recovery of their corresponding connected components as described in the previous subsection.

6 Example Program

In this section we provide a simple program breakdown that contains a sample of each of the connector categories and prove its security. Consider the following program breakdown (Fig. 12 depicts this example).

1. An A-Sec computation f_1 with an input a_1 from Alice and an output oa_1. Bob has to be the garbler of this computation.

2. A 2PC computation f_2 with an input oa_1 from the previous computation, an input b_1 from Bob, an input a_2 from Alice and an output oa_2. Assume that Bob is the garbler of this computation.

3. An A-Sec computation f_3 with an input oa_2 from the previous computation, an input a_3 from Alice , and an output oa_3. Bob has to be the garbler of this computation.

4. A 2PC computation f_4 with an input oa_3 from the previous computation, an input b_2 from Bob , an input a_4 from Alice and an output o as the final output of the program. Assume that Alice is the garbler of this computation.

To securely connect f_1 to f_2 we use the third connector (Fig. 8): blind the output and use OT to check the consistency of the blinding and unblinding values. To connect f_2 to f_3 we use the second connector (Fig. 6): reuse the garbling keys. To connect f_3 to f_4, we use the forth connector (Fig. 9): blind the output and use universal hash to check the consistency of the blinding and unblinding values. Note that based on the order of evaluation (section 5), we have three different connected components. The first one contains only f_1, the second one contains f_2 and f_3, and the last one contains only f_4.

The full protocol and its proof of security for this example are moved to the Appendix due to space limitation. We have used this example as our test case for implementation as described in the next section.

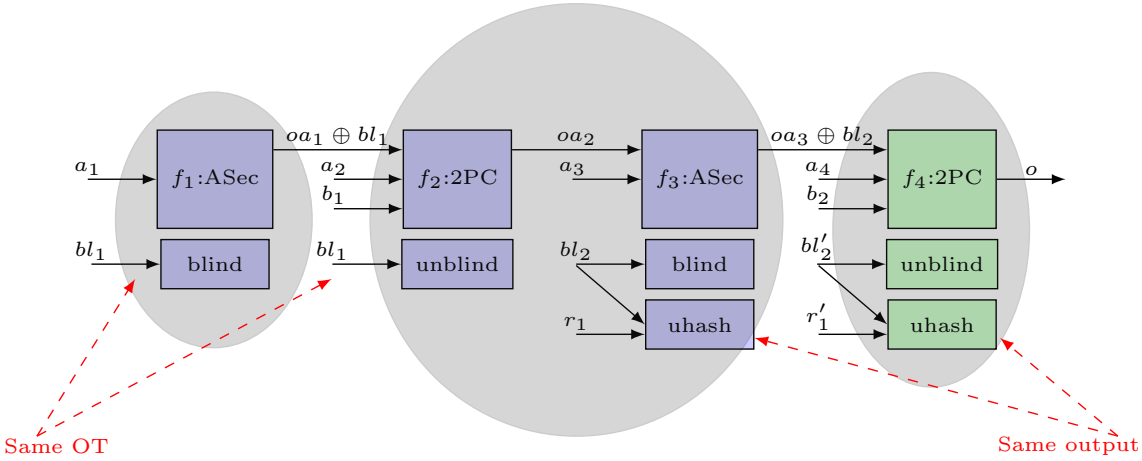

Figure 12: Example program breakdown. Each gray circles indicates a connected component. Computations shown in blue are garbled by Bob and computations shown in green are garbled by Alice.

7 Implementation

Our prototype implementation consists of three stages. In the first stage, the program is written in SCVM language which outputs the program breakdown into smaller pieces and marks each of these pieces as Plain , Both , Alice , or Bob . In the next step, each sub-program is implemented in Frigate [MGC+16] which outputs the corresponding Boolean circuit. Finally, each of these newly generated Boolean circuits are passed to our Java implementation which constructs the connectors, garbles, and evaluates all the circuits. The Java implementation uses the SCAPI [EFLL12] library as a back-end.

The Java implementation of the protocol has five stages. First, the circuits are augmented by adding the necessary blinding and unblinding sub-circuits. We also anticipate the extra inputs required for the universal hash sub-circuits and include them in the augmented circuit, but postpone adding the universal hash function until later in protocol when the appropriate parties are committed to their inputs. Afterwards, the information about the input and output wires of each circuit are read and the parties garble and commit to the garbled keys. In the third stage, we run OT protocol. Next, the parties exchange the hash functions if necessary. At this point, we have the final augmented circuit definition and we proceed to read the new augmented definition to memory. Finally, we loop over the connected components and for each connected component, commit to cut, and continue to garble and evaluate the computations as described in Section 5.

In the experiments results that follows, we have excluded the time it takes to change the circuit definition (e.g. adding blinding sub-circuit) and also the time it takes to read the circuit definition to memory. The main reason for that is that the SCAPI library uses an ASCII text format for circuit definition and as the circuit size grows, parsing this circuit definition files take an unreasonably large amount of time (in some

cases as much as half of the total computation time). One can change the SCAPI library to use a binary circuit representation to speed up this processing cost.

We run the experiments on the example program described in section 6 and depicted in Fig. 12. The example consists of two 2PC and two A-Sec computations. It contains three connected components and uses connectors number 2, 3, and 4 to connect these components together. To compare the efficiency gain of our protocol, we ran each of our experiments in two cases: once for the Mixed computation and once again for Monolithic computation. First, we create the circuits for the example program and count the total number of gates and the total number of the inputs. We then create a monolithic circuit with the same number of total gates and the total inputs size. Note that, when counting the number of gates and inputs, we do NOT count the extra gates and inputs added by our protocol. Finally, we run the example program using the Mixed protocol (Fig. 16 in Appendix) and run the Monolithic program using our modified LR15 protocol of Fig. 15 in Appendix. In the graphs that will follow, we show the efficiency gain as the ratio computed by dividing the time it takes to evaluate the Monolithic computation by the time it takes to compute the Mixed computation.

We performed two experiments both over LAN and WAN on a Linux machine with Intel(R) Xeon(R) CPU E5-2699 v3 2.30GHz and 250GB RAM. In the first experiment, we kept the total input size at 2048 bits and the size of each of the 2PC computations at $256,000$ gates. Note that as per Fig. 12, by total input we mean the sum of $a_1, a_2, a_3, a_4, b_1, b_2$. Moreover, we have broken the total input equally between computations. For example, for a total input of 2048 bits, each of the four computations in Fig. 12 have 512 bits of input. As will be described in the second experiment, the input size of 2048 is a fair value for this particular program. After deciding the input size, we set the size of each of the A-Sec computations to be $0.25, 0.5, \ldots, 64$ times the size of the 2PC computations. In other words, in the first iteration, the total 2PC gates in Mixed computation is $2 \times 256K = 512K$ and the total gates in the A-Sec computation is $2 \times 0.25 \times 256K = 128K$, which results in a total of $640K$ gate for the Monolithic computation. Similarly, in the last iteration, the total 2PC gates in Mixed computation remains $512K$ while the total A-Sec gates are increased to $16M$ gates. Figure 13 shows how the Mixed program performs compared to the Monolithic one in LAN and WAN mode.

As the size of the circuit grows, the dominating cost becomes the cost of garbling and sending the garbled circuits over network. For this reason, we see a great improvement in efficiency when the size of the JKO-based computations are sufficiently larger than the size of the LR15-based computations. In the case of a WAN network with a round-trip delay of 90ms, the efficiency gain is even more due to the longer time spent on sending the garbled circuits.

In the second experiment, we kept the circuit size ratio of 2PC and A/B-Sec the same and in each iteration increased the total input size. This experiment is aimed to examine the effect of the extra inputs that are added by our Mixed protocol. Therefore, we kept the circuit sizes to the bare minimum by setting the circuit size to be is exactly half the input size (e.g. one gate for each two input bits). We started the first iteration of the experiment by a total input size of 128 and doubled the input size in each iteration. In this experiment, the dominating cost are the costs of commitments on inputs, running OT, and checking the consistency of the inputs. As can be seen in Fig. 14 once the input size is large enough to dominate the extra inputs and randomness added by our protocol, the Mixed computation slowly starts to get a small edge over the Monolithic computation due to replication factors (i.e. LR15-based computations perform s times more commitment than their JKO-based computations and the cheating recovery computations have a replication factor of $3s$). Since the network communication is not the bottleneck, the WAN experiments also shows a very small improvement. Note that in this particular example program, for a total input size of 2048, the Mixed mode is still at a disadvantage, but very close to breaking even. That is why we have picked the total input size of 2048 for the first experiment to showcase the importance of the circuit size.

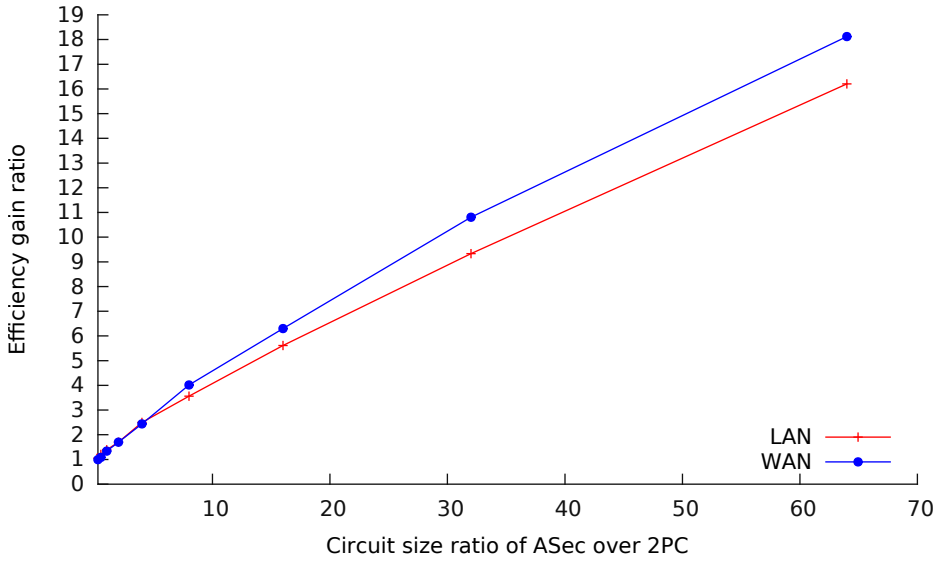

Figure 13: The efficiency gain of the Mixed computation over the Monolithic computation. In this experiment, the total input size is 2048 bits and in each iteration of the experiment, the size of the A-Sec computation is doubled. The total circuit size of the Monolithic computation in the last iteration is 16 million gates.

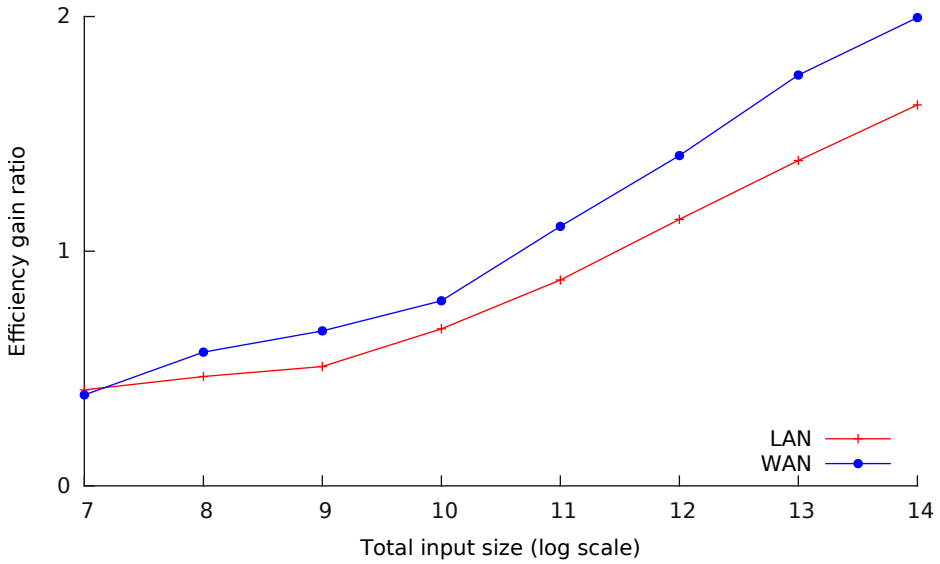

Figure 14: The efficiency gain of the Mixed protocol over the Monolithic. In this experiment, the total input size starts from 128 bits and is increased exponentially to 16,384 bits. The size of A-Sec computations are equal to their 2PC counterparts.

References

[AHMR15] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 702–729. Springer, Heidelberg, April 2015.

[Bau16] Carsten Baum. On garbling schemes with and without privacy. Cryptology ePrint Archive, Report 2016/150, 2016. http://eprint.iacr.org/.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Heidelberg, December 2012.

[EFLL12] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012. http://eprint.iacr.org/.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Oswald and Fischlin [OF15], pages 191–219.

[GG14] Juan A. Garay and Rosario Gennaro, editors. *CRYPTO 2014, Part II*, volume 8617 of *LNCS*. Springer, Heidelberg, August 2014.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Garay and Gennaro [GG14], pages 458–475.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Sadeghi et al. [SGY13], pages 955–966.

[KMW16] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. http://eprint.iacr.org/.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient RAM-model secure computation. In *2014 IEEE Symposium on Security and Privacy*, pages 623–638. IEEE Computer Society Press, May 2014.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 1–17. Springer, Heidelberg, August 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, Heidelberg, May 2007.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, Heidelberg, March 2011.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [GG14], pages 476–494.

[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*, CCS '15, pages 579–590, New York, NY, USA, 2015. ACM.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 458–473. Springer, Heidelberg, April 2006.

[MGBF14] Benjamin Mood, Debayan Gupta, Kevin R. B. Butler, and Joan Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 582–596. ACM Press, November 2014.

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin RB Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. 2016.

[OF15] Elisabeth Oswald and Marc Fischlin, editors. *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*. Springer, Heidelberg, April 2015.

[SGY13] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *ACM CCS 13*. ACM Press, November 2013.

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 386–405. Springer, Heidelberg, May 2011.

[sS13] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Sadeghi et al. [SGY13], pages 523–534.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Oswald and Fischlin [OF15], pages 220–250.

A Modified Protocol of [LR15]

The modified protocol is presented in Fig. 15.

B Full Example Protocol with Proof

This section provides the full protocol for the program described in section 6.

B.1 Preliminaries and Notations

For this protocol we have used the modified protocol of [LR15] as the $\mathsf{2PC}$ protocol (described in Fig. 15). What follows (Fig. 16) is the protocol for secure computation of the program breakdown of section 6 based on the connectors used and the order of evaluation imposed by the connected components.

We use the following notations in the full protocol to make it easier to read and follow. For each computation i, we start by the function f_i corresponding to that computation and show the augmentations made to the computation as follows. The portion of the augmented computation that correspond to the universal hash function is shown as f_i^{uhash}, similarly, f_i^{blind}, $f_i^{unblind}$, f_i^{plain} are used to show the augmented part corresponding to blinding, unblinding, and handling of a plain value (either by hard-coding or creating garbled keys that include truth value). The computation for cheating recovery is shown by $cheat_j$, where j is used to indicate the jth connected component.

B.2 Full protocol

We proceed to the full protocol for our mixed-2PC program described in Fig. 16 and start by addressing general issues regarding integrating different components of computations together.

First, is the issue of having the parties original inputs fed into different computations which would cause checking input consistency across different computations troublesome. In this case we create dummy computations that behave as identity functions. The inputs that need to be passed to more than one computation are first fed into these identity computations and then the output of these identity computations are connected to the desired computations based the type of the connector that is needed to connect the two computations. Therefore, when dealing with the input consistency, we always deal with "one" computation. If the input is fed to a $\mathsf{2PC}$, the consistency of the input is checked as described in LR15. If the input is fed to $\mathsf{A/B\text{-}Sec}$, since we only have one garbled circuit, no input consistency check is required. For other inputs to the computation, the consistency is either checked by checking the output of the universal hash function or by using the same OT between the two computations.

Second, is the issue of extracting the parties inputs in the simulation proofs. Consider the ideal world model where the parties send all their inputs to the ideal functionality, the ideal functionality computes each of the subcomputations and sends the appropriate outputs to each party. In this model, the parties choose and fix their inputs at the beginning of the computation, before receiving any output. Looking at the proof of security of the [LR15] protocol, the garbler's inputs are extracted right before the evaluation phase. This implies that when using the modified LR15 as the $\mathsf{2PC}$ back-end in our protocol, the garbler of a connected component can choose his input after learning about the output of the previous connected components. To fix this issue, we require that at the beginning of protocol, the parties commit to their chosen garbled inputs for the computation in which they are the garbler. This results in a clear, easy to describe, and easy to simulate protocol at the cost of a small overhead (commitments on garbler's garbled inputs). Of course, with this change we make a small adjustment to the LR15 protocol and check these new commitments against the commitments used in input consistency check (i.e. $F_{\mathsf{ExCom\Delta ZK}}$). We stress that the cost of these changes are insignificant compared to the efficiency gain of reducing the amount of garbled circuit as stated in section 7.

Setup: Let s be the statistical security parameter and k be the computational security parameter. Let x be the Garbler's private input and y be the Evaluator's private input. Also, let D be the public input of the Garbler. The parties decide $C(x, y)$ which computes $f(x, Ey)$ and $C'(x, D, d)$ which computes $f(x, D, E'd)$ where it returns x if $D = d$ and returns 0 otherwise. E, E' are probe resistant matrices. Let $\mathsf{label}(gc_i, j, b)$ represent the garbled key corresponding to the jth wire of garbled circuit gc_i, where b represent the truth value. Moreover, let $\mathrm{IN}(.)$ and $\mathrm{OUT}(.)$ represent the set of input and output wires.

Garbling input & output wire keys: For $i = 1, \ldots, s$, the garbler chooses a random value $seed_i \in_R \{0,1\}^k$ and uses $\mathsf{PRF}_{seed_i}(.)$ to generate all the randomness required for this step and step "Garbling".

1. For input wires corresponding to x, y, d generate the garbled keys normally.
2. For input wires corresponding to D and the output wires of circuit C, C', generate the garbled keys such that the firs bit is the truth value (i.e. $b||r$ such that $r \in_R \{0,1\}^{k-1}$ and $b = 0$ if this garbled key represent the value 0).
3. The garbler chooses the permutation bit-string $m_i \in_R \{0,1\}^{|x|}, i \in [s]$ and $m'_i \in_R \{0,1\}^{|x|}, i \in [3s]$.
4. The Garbler computes the commitment $\mathsf{Com}(\mathsf{label}(gc_i, j, b))$ for $b \in \{0,1\}$ and $j \in \mathrm{IN}(C) \cup \mathrm{IN}(C') \cup \mathrm{OUT}(C)$ and permutes the input commitments based on the corresponding m_i and m'_i.

Oblivious Transfer for y: The parties run probe-resistant OT protocol in which the Evaluator inputs y and the garbler inputs the garbled keys for input wires corresponding to y. The Evaluator learns the garbled keys for y in all garbled circuits of C along with the corresponding decommitment.

Commit to "cut": The Evaluator

1. chooses a random string σ of length s such that each bit of σ is chosen to be 1 with probability $1/2$. The set of indexes E is defined to be such that if the jth bit of σ is 1, then j is in E with the constrained that $E \neq [s]$.
2. chooses σ' of length $3s$ at random such that exactly $3cs$ bits are 1, where $0 < c < 1$ is a constant defined by the particular $\mathsf{2PC}$ protocol chosen for computing C'. The Evaluator sends $\mathsf{Com}(\sigma), \mathsf{Com}(\sigma')$.

Garbling: In this step the garbled circuits for both computations are created and sent. For $i = 1, \ldots, s$, the Garbler keeps using the appropriate $PRF_{seed_i}(.)$ to derive the randomness for this step.

1. The Garbler creates s independent garbled circuits from C (i.e. $gc_1, \ldots, gc_s$) and $3s$ independent garbled circuits from C' (i.e. $gc'_1, \ldots, gc'_{3s}$).
2. The Garbler computes $\mathsf{ExtractCom}(\mathsf{seed_i}), i \in [s]$.

Run $F_{\mathsf{ExCom\Delta ZK}}$:

1. The parties run $F_{\mathsf{ExCom\Delta ZK}}$ where the Garbler inputs $m_i, i \in [s]$ and $m'_i, i \in [3s]$ and the Evaluator inputs the set B_1 (i.e. only a single run) such that B_1 corresponds to the set of garbled circuits chosen for evaluation as per σ for C and σ' for C'.
2. The Evaluator learns the set of Δ_i for both C and C'.
3. The Garbler learns the choice of B_1.
4. The Evaluator decommits to σ and σ' and the garbler confirms that the set B_1 matches the decommitment. Note that the Evaluator does not decommit to the seeds yet.

Figure 15: 2PC - cheating recovery - single execution

Evaluate C:

1. The Evaluator already knows the garbled values for y and its decommitments.
2. The Garbler sends $x_1 = x \oplus m_1$ were m_1 is the first permutation value in the set of evaluation circuits. Knowing Δ_i and x_1, the Evaluator can compute $x_i = x \oplus m_i$ on its own.
3. For $i = 1, \ldots, s$, the Garbler sends garbled keys and decommitments for the input wires corresponding to x_i.
4. The Evaluator checks that all the decommitments are valid and correspond to x_i.
5. The Garbler chooses a value $D \in_R \{0,1\}^k$.
6. For $j \in \mathrm{OUT}(C)$,
   - The Garbler chooses random value $R_j \in_R \{0,1\}^k$.
   - The Garbler sends the encryption of R_j and $R_j \oplus D$ under the garbled output keys. I.e. he sends $(\mathsf{label}(gc_i, j, 0) \oplus R_j, \mathsf{label}(gc_i, j, 1) \oplus R_j \oplus D)$.
7. The Garbler computes $\mathrm{H}(D)$.
8. The Evaluator evaluates gc_i and obtains an output key $out_{i,j}$ and uses it to decrypt the corresponding R_j and $R_j \oplus D$ values.
9. If all the output keys of a wire j decrypt to the same value, set z to be that output (but do not halt).
10. If they decrypt to different value, the Evaluator checks their XOR and verifies it against $\mathrm{H}(D)$. If, successful, the Evaluator sets d to be the result of the XOR.

Evaluate C':

1. The parties run OT protocol on d.
2. The Garbler provides D in plain and opens the corresponding commitments. Due to the way D is garbled, checking the consistency of the garbled values of D is immediate.
3. Similar to before, the Garbler provides $x'_1 = x \oplus m'_1$ and opens the commitments for garbled values of $x'_i = x \oplus m'_i$.
4. Moreover, the Garbler opens the commitment on the output garbled keys of circuit C and reveals D. The Evaluator check if they are all correct and that the output garbled keys decrypt to the correct R_j and $R_j \oplus D$ values.
5. The parties check and evaluate C' as per specific $\mathsf{2PC}$ protocol chosen and the Evaluator accepts the majority output as $\hat{x}$.
6. If the Evaluator has indeed obtained a valid d, he computes $z = f(\hat{x}, y)$.

Opennig/Checking C:

1. For $i \notin E$ (calculated from σ), the Garbler sends the decommitmens on $seed_i$.
2. the Evaluator checks the garbled circuits and all the input and output commitments and verifies that they are all correct.

Output:

1. If all evaluation circuits of C return the same output z, then the Evaluator outputs z.
2. If the Evaluator recovers the correct d, he outputs $z = f(\hat{x}, y)$.
3. Otherwise, let gc_i be the computation were all the garbled output values were valid (matched against the commitments), then the Evaluator outputs the result of gc_i as z.
4. If required, the Evaluator reveals the garbled output labels corresponding to z as proof that z is indeed the output of the computation.

Figure 15: 2PC - cheating recovery - single execution

Order of initial operations. We start by augmenting all the computations based on the connectors. In other words, the parties add blinding and unblinding subcomputations. The parties also augment the computation by creating the extra input wires needs for the universal hash functions (although, they do not create the universal hash function at this point). Moreover, the parties create the cheating recovery computation for each connected component. Next, the garbler of each computation chooses the garbled input and output keys as instructed by the corresponding connector.

After that, we run the OT on all the inputs (both for the original inputs and for the extra inputs added by the connectors). Similarly, the garbler of each computation commits to his garbled inputs. Finally, the parties go over each connected component and perform the necessary steps.

B.3 Proof

In the ideal world, the ideal functionality has access to the program breakdown. The parties send all their inputs to the ideal functionality. The corrupted party will then receive all the outputs that he is allowed to see from each of the subcomputations. To enable aborting at any time in protocol, for each subcomputation, the ideal functionality waits to receive an acknowledgment from the corrupted party. If it receives the acknowledgment, it reveals the appropriate values to the honest party. Otherwise, it sends the honest party $\perp$ for all remaining subcomputations.

Assume a simulator $\mathcal{S}$ (i.e. the corrupted party) in the ideal world. The simulator runs the protocol of Fig. 16 internally. Through simulation, we prove that given only the inputs of the malicious party (controlled by an adversary $\mathcal{A}$), the simulator $\mathcal{S}$ can generate a view of $\mathcal{A}$ (joint distribution of all messages received by $\mathcal{A}$) in real world that is indistinguishable from that of the ideal world (joint distribution of the inputs and outputs the simulator). We assume that if a party is honest, it will remain honest in all the subcomputations, regardless of the role he/she is taking.

We show the output of the ideal world with z: if the output is named out in the real world protocol, we use z_{out} to show the corresponding output obtained from the ideal functionality.

Alice is corrupted.

For an adversary $\mathcal{A}$, the simulator $\mathcal{S}$

- Performs steps 1 and 2 and extract $\mathcal{A}$'s inputs $a_1, a_2, a_3, bl_1, bl_2$ through the corresponding OTs. The simulator also extract a_4, bl'_2 through the corresponding commitments.
- Sends a_1, a_2, a_3, a_4 to the ideal functionality and obtain $z_{oa_1}, z_{oa_2}, z_{oa_3}, z_o$.
- Chooses random values for b_1, b_2.
- Performs step 3 - 6 honestly and aborts if the revealed output of f_1 is not equal to $z_{oa_1} \oplus bl_1$.
- Performs steps 7 and 8 and extracts $\mathcal{A}$'s choice of cut-and-choose, σ_2, for f_2 in step 8.
- In step 9, $\mathcal{S}$ garbles f_2 and $cheat_2$ according to the σ_2 and z_{oa_2} in a standard way (i.e. garbling the circuits chosen to be opened honestly and garbling the rest of the circuits to always output z_{oa_2}). The simulator garbles f_3 honestly.
- Performs steps 10 - 15, and aborts if the output of f_3 is not equal to $z_{oa_3} \oplus bl_2$.
- Performs steps 16 and 17 and aborts if all the garbled circuits it has chosen for checking are correct and all the garbled circuits for evaluation are incorrect.
- Performs steps 18 - 20 and aborts if $bl_2 \neq bl'_2$ (extracted in the first step of simulation).

Initial steps:

Since each original input (i.e. $a_1, a_2, a_3, a_4, b_1, b_2$ is connected to only one computation, no extra identity computation is necessary).

1. $Construct$
   - f_1^{blind} is decided as per the 3rd connector.
   - $f_2^{unblind}$ is decided as per the 3rd connector.
   - f_3^{blind} is decided as per the 4th connector.
   - $f_4^{unblind}$ is decided as per the 4th connector.
   - $cheat_2$ is decided as the cheating recovery for the second connected component. In case of cheating, it would reveal b_1.
   - $cheat_3$ is decided as the cheating recovery for the third connected component. In case of cheating, it would reveal a_4, bl_2'.
   - Augmentations adding universal hash functions (i.e. f_3^{uhash} and f_4^{uhash}) will happen later in the protocol as instructed by the connector in use.
2. $GarbIO\&OT$
   - Alice chooses, random blinding value bl_1. She also chooses random blinding and unblinding values bl_2, bl_2' such that $bl_2 = bl_2'$.
   - Bob chooses universal hash function H but does not sends it to Alice.
   - The parties choose garbled keys for input and output wires of all computations and augmented computations according to the connector in use.
   - The plain values (i.e. the blinded output values), are handled by choosing the garbled input keys to include the truth value.
   - For all inputs (original inputs, or the extra blinding/unblinding ones) that belong to the Evaluator of a computation, they run OT. The garbler also commits to his garbled inputs.
     - a_1: batch committing OT.
     - a_2: prob-resistant OT.
     - a_3: batch committing OT.
     - b_2: prob-resistant OT.
     - bl_1: the same prob-resistant OT for both f_1^{blind} and $f_2^{unblind}$
     - bl_2: prob-resistant OT.
     - a_4, b_1, bl_2': commit to garbled input

Connectors in use:

The first connected component includes only f_1. The second connected component includes f_2 and f_3. The third connected component includes only f_3. Each connector used is marked with a symbol for better readability in the next steps of the protocol.

- $\star : f_1 \rightarrow f_2$: Use the 3rd connector (Fig. 8) to connect the first to the second connected component.
- $\diamond : f_2 \rightarrow f_3$: Use the 2nd connector (Fig. 6) for internal connection in the second connected component.
- $\odot : f_3 \rightarrow f_4$: Use the 4th connector (Fig. 9) to connect the second to the third connected component.

1st connected component (contains f_1, involves $\star$):

Since we only have one computation, no internal connector is necessary. Therefore, as described in section 5, we only perform the first half of the connector that is used to connect this connected component to the next.

3. $\star : [(f_1, f_1^{blind})]_{Garble\&Commit}$: Bob garbles the augmented computation and sends it to Alice.
4. $\star : [(f_1, f_1^{blind})]_{EvalMain}$: Alice evaluate and commits to garbled output $oa_1 \oplus bl_1$.
5. $\star : [(f_1, f_1^{blind})]_{Open}$: Bob opens the garbled circuit and also opens the OT for bl_1 for this computation.
6. $\star : [(f_1, f_1^{blind})]_{Reveal}$: If the checks pass, Alice opens the output commitments. Both parties learn the plain output of the computation.

Figure 16: Example full protocol.

2nd connected component (contains f_2, f_3, involves $\star, \diamond, \odot$):

This connected component uses the 2nd connector (Fig. 6) as internal connector for f_2 and f_3 it also contains the second half of $\star$ (used to connect f_1 to f_2) and the first half of $\diamond$ used to connect f_3 to f_4. The steps are broken into 4 phases for better readability: the garbling phase, the evaluation phase, the cheating recovery phase, and the opening phase.

—The garbling phase:

7. $\odot : [(f_3^{uhash})]$: Since Alice is already committed in step $GarbIO\&OT$, Bob sends the universal hash hash function H that he has chosen to Alice.
8. $\star, \diamond : [(f_2, f_2^{unblind})]_{Cut}$: Alice commits to choice of cut-and-choose for f_2.
9. $\star, \diamond, \odot$: $[(f_2, f_2^{unblind}, cheat_2), (f_3, f_3^{blind}, f_3^{uhash})]_{Garble\&Commit}$: Bob garbles the augmented computations and the cheating recovery computation and commits to the garbled inputs and outputs accordingly.
10. $\star, \diamond : [(f_2, f_2^{unblind}, cheat_2)]_{GarblerInput}$: Bob proves the consistency of b_1 passed to f_2 and the cheating recovery computation.

—The evaluation phase:

11. $\star, \diamond : [(f_2, f_2^{unblind})]_{EvalMain}$: Bob sends the decommitment for $oa_1 \oplus bl_1$ and Alice verifies them against their truth value. Bob also sends his garbled inputs and Alice checks them as instructed by $F_{\mathsf{ExCom}\Delta\mathsf{ZK}}$. Alice also checks these value against the commitments on the garbled inputs given in step two of the protocol. Alice already knows the garbled values for a_2 from the OT.
12. $\diamond, \odot : [(f_3, f_3^{blind}, f_3^{uhash})]_{EvalMain}$: Alice has already obtain the garbled inputs for a_3, bl_2 using OT.

—The cheating recovery phase:

13. $\star, \diamond : [(cheat_2)]_{EvalCheatRecov}$: As described in Fig. 8. If successful, Alice learns b_1 and can compute the output of f_2 and f_3.

—The opening phase:

14. $\star, \diamond, \odot$: $[(f_2, f_2^{unblind}), (f_3, f_3^{blind}, f_3^{uhash})]_{Open}$: Alice verifies the garbled circuits, the commitments, and the rest of the OTs for bl_1 (from the $\star$).
15. $\star, \diamond, \odot : [(f_2, f_2^{unblind}), (f_3, f_3^{blind}, f_3^{uhash})]_{Reveal}$: Only the $oa_3 \oplus bl_2$ and $\mathrm{H}(bl_2)$ are revealed to Bob.

Figure 16: Example full protocol.

3rd connected component (contains f_4, involves $\odot$):

Since we only have one computation, no internal connector is necessary. Therefore, as described in section 5, we only perform the second half of the connector that is used to connect the previous connected component to this one.

16. $\odot : [(f_4, f_4^{unblind}, f_4^{uhash})]_{Cut}$: Bob commits to his choice of cut-and-choose for f_4.
17. $\odot : [(f_4, f_4^{unblind}, f_4^{uhash}, cheat_3)]_{Garble\&Commit}$: Alice garbles the augmented f_3 and commits to garbled input and output wire keys.
18. $\odot : [(f_4, f_4^{unblind}, f_4^{uhash}, cheat_3)]_{GarblerInput}$: Alice proves the consistency of a_4, bl_2' passed to f_4 and the cheating recovery computation.
19. $\odot : [(f_4, f_4^{unblind}, f_4^{uhash})]_{EvalMain}$: Alice sends decommitments on input wires of $oa_3 \oplus bl_2$ and Bob verifies the truth value. Alice sends her garbled inputs based on both F_{ExComAZK} and the input commitments of [sS13]. Bob has already received the garbled inputs for b_2 using OT.
Bob verifies $\mathrm{H}(bl_2')$ is equal to $\mathrm{H}(bl_2)$ of the previous computation.
20. $\odot : [(cheat_3)]_{EvalCheatRecov}$: As described in the 4th connector 9. If successful, Bob learns a_4 and bl_2' and he already knows $oa_3 \oplus bl_2$ and since he has checked that $\mathrm{H}(bl_2) = \mathrm{H}(bl_2')$, he can compute $f_4(oa_3 \oplus bl_2' \oplus bl_2, b_2, a_3)$ and obtain o the result of the computation.
21. $\odot : [(f_4, f_4^{unblind}, f_4^{uhash})]_{Open}$: Bob verifies the augmented f_4.
22. $\odot : [(f_4, f_4^{unblind}, f_4^{uhash})]_{Reveal}$: If all checks pass, Bob reveals o and proves to Alice that he is not lying about o by also sending the garbled output values corresponding to o.

Figure 16: Example full protocol.

- Performs steps 21 and 22 and outputs whatever $\mathcal{A}$ outputs.

The simulation differs from the real evaluation in 6 major ways.

1. In step 6, if the output of f_1 is different from the $z_{oa_1} \oplus bl_1$. The indistinguishability argument of this case is slightly different from that of JKO. In JKO, the simulator (i.e. the verifier) would check the validity of the witness (in our case Alice's input) and would abort if the witness was invalid. In other words, since the output of the function was public, the simulator would apply the function on the extracted witness and compare the result with publicly known output of the computation. The work of JKO then reduced the problem of receiving a different "real" output than the computed "ideal" output to breaking the privacy property of the garbled circuit.

   In our case, the output is not public, therefore the simulator instead compares the ideal result (i.e. the result obtained from applying the function on the extracted input/witness) with the actual output of the computation. We also use the same argument that having the same input passed to an ideal function and to a correctly constructed garbled circuit and yet obtaining different outputs, breaks the privacy property of the garbling scheme.

   Note that the value bl_1 does not affect the above argument since due to the security of the OT protocol, the adversary has obtained the garbled value for bl_1 and by the authenticity property of the garbling scheme, she cannot change that value.

2. In step 9, where the simulator garbles the circuits chosen for evaluation to always output the same output, regardless of their garbled input. The indistinguishability argument of this case reduces to the privacy property of the garbling scheme.

3. In step 9, where the simulator garbles the cheating recovery circuits to always output zero. Since the simulator is not cheating, the adversary cannot obtain different garbled outputs (due to authenticity of the garbling scheme) and therefore the result of the cheating recovery computation will always be zero.

4. In step 15, if the output of f_3 is different from $z_{oa_3} \oplus bl_2$. As before, this case is also indistinguishable from the real execution.

5. In step 17, if all the evaluation circuits are incorrect and all the open circuits are correct. The adversary can distinguish the simulator, but this only happens with negligible probability in the statistical security parameter.

6. In step 20, aborting if the $bl_2 \neq bl'_2$ (i.e. regardless of the actual output of the hash functions). In this case, the adversary is committed to bl_2 using OT and she is committed to bl'_2 using a perfectly binding commitment and therefore cannot change the inputs after learning about H. Moreover, the output of H are not revealed to the adversary until the correctness of the garbling is checked. Therefore, the event where $\mathrm{H}(bl_2) = \mathrm{H}(bl'_2) \wedge bl_2 \neq bl'_2$ can happen if it is a collision or if the adversary has cheated in cut-and-choose. Both of which happen with negligible probability in s.

Bob is corrupted.

For an adversary $\mathcal{A}$, the simulator $\mathcal{S}$

- Performs steps 1 and 2. Extract $\mathcal{A}$'s inputs b_2 through the corresponding OT. Extract b_1 through the corresponding commitment.
- Sends b_1, b_2 to the ideal functionality and obtain z_o.
- Chooses random values for $a_1, a_2, a_3, bl_1, bl_2 = bl'_2$.
- Performs steps 3 - 16 honestly.
- Extracts $\mathcal{A}$'s cut-and-choose choices, σ_4, in step 17 and garbles f_4 and $cheat_3$ based on σ_4 and z_o in the standard manner.
- Performs the rest of the steps honestly and output whatever $\mathcal{A}$ outputs.

The simulation is different from the real evaluation in 4 major ways:

1. In steps 4 and 13 where the simulator commits to the output of an A-Sec computations. In these cases, if the adversary sends garbled circuit or garbled keys that are incorrect, the simulator will commit to $\perp$ while a real party (Alice) would commit to a different garbled output. But these case are caught in opening phase and as a result the commitments are not opened. Therefore, the indistinguishability argument reduces to the hiding property of the commitment scheme.

2. In step 9, if all the garbled circuits chosen for evaluation are incorrect and the rest are correct. As before, with negligible probability the adversary can distinguish the simulator.

3. In step 17, where the simulator garbles the evaluation circuits to always output z_o. As before, this reduces to the privacy property of the garbling scheme.

4. In step 17, where the simulator garbles the cheating recovery circuit to always output zero. The indistinguishability argument is as before.