Better Two-Round Adaptive Multi-Party Computation*

Ran Canetti† Oxana Poburinnaya‡

Abstract

The only known two-round multi-party computation protocol that withstands adaptive corruption of all parties is the ingenious protocol of Garg and Polychroniadou [TCC 15]. We present protocols that improve on the GP protocol in a number of ways. First, concentrating on the semi-honest case and taking a different approach, we show a two-round, adaptively secure protocol where:

- Only a global (i.e., non-programmable) reference string is needed.
- The communication complexity depends only on the size of RAM description of the evaluated function (and not on its circuit size). The work of each party depends on RAM complexity of the function.
- Even not well-formed randomized functionalities can be evaluated securely.
- Only polynomially-secure indistinguishability obfuscation for circuits and injective one way functions are assumed.

Second, we modify the GP protocol to have only RAM complexity even in the case of Byzantine corruptions. For this we construct the first statistically-sound non-interactive Zero-Knowledge scheme with RAM complexity.

*Research Supported by the NSF MACS Frontier project and NSF Algorithmic Foundations grant 1218461.

†Tel-Aviv University and Boston University. canetti@bu.edu. Supported in addition by the Check Point Institute for Information Security, ISF grant 1523/14.

‡Boston University. oxanapob@bu.edu.

Contents

1 Introduction

Adaptive security of protocols, namely security against an adversary that decides whom to corrupt adaptively during the execution of the protocol, has been an ongoing focus in cryptography. Indeed, adaptive security better captures real life adversaries, which can often make adaptive corruption choices.

Two cases which are of particular importance in this setting are (a) the case where no data erasures are possible, hence the adversary gets to see all the past internal states of a corrupted party, and (b) the case where all parties are eventually corrupted. Indeed, while for static corruptions the case of all parties being corrupted is uninteresting, for adaptive corruptions the case of all parties being eventually corrupted allows arguing about the security of protocols for computing randomized functions, where the randomness remains hidden even given the output values themselves. It also allows arguing about the security of other, uncorrupted parties in a larger system which uses our protocol. Furthermore, the combination of these properties allows demonstrating leakage tolerance properties even when all parties may leak some side-channel information on their local computations [BCH12]. We call protocols that are secure in this setting fully adaptive.

Constructing fully adaptive protocols is a significant challenge. The difficulty here is that the adversary eventually sees all the inputs and random choices of the parties, and yet security of the output and the computational process should be maintained. Indeed, such protocols with constant number of rounds appeared only recently [CGP15, DKR14, GP14]; among these protocols, only [GP14] is a multiparty protocol with optimal (i.e. two) number of rounds.

The focus of this work is to construct better two-round, fully adaptive protocols for general multi-party computation. Our improvements span a number of security, functionality, and efficiency aspects. We start by presenting and discussing some of these aspects.

Global common reference string. In a common reference string (CRS) model, all parties have an access to a string, generated in advance by a trusted entity (which doesn't need to participate in the protocol). In the *global, or non-programmable,* CRS model the simulator in the ideal world is required to simulate the execution with a given CRS; it cannot generate the CRS itself (for instance, it cannot generate the CRS for which it knows the trapdoor). Consequently, while it is easier to construct and prove security of protocols in the local CRS model, the global CRS model makes significantly weaker trust assumptions on the reference string and its generation process. In particular, a global CRS can be known globally and used by all protocols in the system without any prior coordination. In contrast, to preserve security of a protocol that was proven secure in the local CRS model within a larger system, one has to make sure that no other protocol in the system will ever use that same CRS, either inadvertently or via malicious protocol design. See e.g. [CDPW07] for more discussion.

Randomness-hiding functionalities. Assume that a group of parties wants to use our protocol as a subprotocol to jointly generate an obfuscated program, which they need in a larger system. They would like to preserve security of the obfuscated program even when everybody in this group is corrupted (which could be important for remaining honest parties in the larger protocol). Note that this functionality (of obfuscating the program) is randomized. Usually protocols are designed to compute deterministic functionalities, and if randomized functionality needs to be computed, then the functionality should derive randomness for

computation from inputs of parties in some deterministic way [1]. However, with this approach the adversary corrupting everybody learns randomness of the function (e.g. random coins of obfuscation), and thus no security is left.

A similar problem exists when adaptive security is used to argue leakage resilience ([BCH12]): assume parties are computing a randomized functionality, and the adversary decides to leak 1 bit of each party's randomness. With the common approach for computing randomized functionalities described above, the simulator from [BCH12] cannot simulate such leakage, since the argument from [BCH12] requires that the simulator could potentially simulate *full randomness of each corrupted party* [2] (even though the adversary actually sees only a single bit of randomness of each party).

The common approach to mitigate these shortcomings is to simply relax security requirements and define a weaker functionality, which reveals all its internal random coins when everybody is corrupted[3]. Such functionalities are called *adaptively well-formed* [CLOS02]. However, functionalities which still have secret internal state (e.g. randomness of the obfuscation) hidden from the adversary even when all inputs and outputs are known (sometimes called adaptively ill-formed ones) allow for stronger security guarantees, such as security of obfuscated program in the above example. We call such functionalities *randomness-hiding.* Note that secure multi-party computation for such functionalities in the plain model is impossible, since after corrupting everybody the adversary learns all internal state of the execution.

Computation and communication complexity. The majority of existing protocols assume that the function is represented as a circuit. This means that the work of parties and, in some cases, the length of communication both depend on the size of a circuit to be computed. Given that Turing machines and RAM machines may have significantly more efficient parameters than circuits, building MPC protocols which use the advantage of more efficient models of computation is an important task. (In particular, in the case of RAM computation that does not necessarily need to access all the input, the gap could be exponential.)

Multiparty computation can benefit from the RAM model in several ways. As the most extreme example, consider the task of searching a word in n exponentially large databases shared by parties: unlike circuit or TM computation, the RAM computation could be logarithmic in the size of databases. Even if there is no exponential advantage, still many tasks (such as indexing, i.e. on input (DB, i) retrieving $DB[i]$) are way more efficient in the RAM model then in the circuit or TM model. Finally, if parties are willing to trade some security for efficiency, then they can obtain efficiency close to the input-specific running time (rather than worst-case running time)[4]. For instance, let's say there is a database with medical data, and a group of researchers is interested in average age of persons satisfying some sparse property P (say, having rare medical condition). If these researches don't care about hiding P, then they can compute the average fairly efficiently, with running time comparable to the number of entries satisfying P. However, if P cannot be made public, then need to run a protocol with P being their secret input; this immediately makes their running time worst-case (for all possible P), which is comparable to the size of the database. If these researches are willing to sacrifice some security to gain efficiency (for instance, if others are allowed to

[1] For instance, parties can choose randomness r_i, make it part of their input, and evaluate the functionality $F((x_1, r_1), \ldots, (x_n, r_n)) = f(x_1, \ldots, x_n; \bigoplus r_i)$.

[2] Or, to be more precise, that there exist a translation function which maps ideal world internal state into real world internal state.

[3] In a case of leakage, even weaker functionality is needed.

[4] Recall that the security of MPC requires that no information about inputs of parties is leaked. Running time of a program M on input x could potentially leak information about x. Therefore if full security is needed then programs should necessarily work as long as their worst-case running time, even if computation on this particular input is short.

learn that P is a rare disease, but cannot learn which one), then they can perform very efficient computation (like in the first case), while still having meaningful security guarantees. Another example is a simplex algorithm for linear programming, which has exponential worst-case complexity, but runs in polynomial time on most inputs.

1.1 Our results

Semi-honest setting. Our main result is the first two-round MPC protocol with global (non-programmable) CRS, which is secure against adaptive semi-honest corruption of all parties. The protocol allows to securely compute even randomness-hiding functionalities, and furthermore, even in this case it guarantees leakage resilience (for the discussion on why this is usually not the case, see the paragraph about randomness-hiding functionalities in the first part of the introduction). In addition, the protocol is RAM-friendly, i.e. the amount of communication in our protocol only depends on the RAM size of a function, not on its circuit size, and the work of each party which obtains the output is proportional to RAM complexity of the function. We assume only polynomially secure IO and injective OWFs.

Theorem 1. *Assuming injective one way functions and indistinguishability obfuscation for circuits, there exists a two-round multiparty protocol with global CRS for computing any randomized functionalities, even randomness-hiding ones. The protocol is adaptively secure against honest-but-curious corruptions of possibly all parties, with oblivious simulation. Its communication complexity depends on $\lambda, \{|x_i|\}_{i=1}^n, y, |f|_{\mathsf{RAM}}$ (logarithmic parameters omitted), and time and space of every party depends on λ, $\{|x_i|\}_{i=1}^n, y, |f|_{\mathsf{RAM}}$, and time or space needed to evaluate RAM $f(x_1, \ldots, x_n)$ in the worst case.*

Our result improves the state of the art in a number of ways. In particular, this is:

- The first 2-round fully adaptive MPC with a global CRS;[5]
- The first 2-round fully adaptive MPC which doesn't require subexponential security of iO;
- The first 2-round fully adaptive MPC which supports all (even randomness-hiding) functionalities, and which therefore is fully leakage resilient.

Malicious setting. As an additional result, we show how to make the protocol of [GP14] RAM-efficient: namely, we construct RAM-efficient statistically-sound non-interactive zero-knowledge proofs, and then plug this NIZK into the protocol of [GP14].

Theorem 2 ([GP14])**.** *Assuming the existence of RAM-efficient statistically sound NIZK, subexponentially secure iO for circuits, and one way functions, there exists a two-round multiparty protocol with local CRS adaptively secure against malicious corruptions of possibly all parties. Its communication complexity depends on $\lambda, \{|x_i|\}_{i=1}^n, y, |f|_{\mathsf{RAM}}$ (logarithmic parameters omitted), and time and space of every party depends on λ, $\{|x_i|\}_{i=1}^n$, $y, |f|_{\mathsf{RAM}}$, and time or space needed to evaluate RAM $f(x_1, \ldots, x_n)$ in the worst case.*

[5] We underline that the approach of [GP14] requires a local CRS even in the honest-but-curious setting.

RAM-efficient statistically sound NIZK. We construct the first RAM-efficient NIZK with statistical soundness, assuming statistically-sound NIZK for circuits (which can be obtained from trapdoor permutations) and a RAM-efficient garbling scheme (which can be built from iO and OWFs [CH16]) :

Theorem 3. *(Informal) Assuming statistically sound non-interactive zero knowledge (NIZK) for circuits and a succinct garbling scheme for* RAM*, there exists a NIZK for RAM, where the work of the prover and the size of the proof depends on* $|R|_{\mathsf{RAM}}$*, and the work of the verifier depends on the RAM complexity of* R *(where* $R(x, w)$ *is a relation which defines the language for the proof).*

We note that our succinct NIZK is useful also in other settings. For instance, in the two-round protocol of Garg et. al. ([GGHR14]) the parties exchange obfuscated programs which compute next message functions (of some underlying many-round protocol) together with a proof that the computation was done correctly. If the underlying protocol has number of rounds proportional to the RAM complexity of the function (say, the protocol by Damgard et. al., [DMN11]), plugging our RAM-efficient NIZK makes [GGHR14] protocol RAM-efficient.

1.2 Related work

Fully adaptively secure protocols. Until now, only three constant-round fully adaptively secure protocols were known. [CGP15] is a two-round protocol for two-party computation; [DKR14] is an MPC protocol, but requires 4 rounds; both protocols have global CRS and allow to compute randomness-hiding functionalities. [GP14] is a two-round MPC protocol secure against malicious adversaries; thus their reference string is necessarily local[6]. Their protocol doesn't support randomness-hiding functionalities.

All three protocols are based on the circuit representation of the function: namely, the core part in both [CGP15, DKR14] are Yao garbled circuits[7]. The protocol of [GP14] requires a statistically-sound NIZK for the statement $f(x_1, \ldots, x_n) = y$, and prior to our work such proofs required verification time proportional to the size of the circuit.

In addition, [CGP15, GP14] require subexponentially-secure iO.

RAM-efficient protocols. Existing protocols for (even static) RAM MPC follow one of the two approaches. The work of Boyle et al ([BCP15]) shares a paradigm of Damgard et al ([DMN11]) which instructs parties to jointly evaluate steps of a RAM CPU; this approach results in number of rounds proportional to the number of CPU steps needed to compute a function.

The other approach, introduced by Ishai and Kushilevitz ([IK02], [AIK06]), requires parties to jointly evaluate a randomized encoding of the function and input and then locally compute the output of this randomized encoding. Thus, plugging a RAM-efficient garbling scheme ([CHJV15, CH16]) into known constructions results in statically-secure RAM-efficient protocols; for full adaptive security the underlying protocol must support randomness-hiding functionalities.

[6] We note however that merely using their protocol in the semi-honest case doesn't allow for a local CRS: their approach requires proving statements to an obfuscated program, which requires NIZK (and therefore a local CRS) even in the honest-but-curious case.

[7] Which cannot be easily switched to the garbling scheme for RAM. For instance, in both protocols the underlying garbling scheme should support bit-by-bit garbling of an input. [DKR14] makes even further use of the actual construction of garbled circuits.

Constant round adaptively secure RAM-efficient protocols. Combining several existing techniques, it is possible to construct adaptively secure protocols for RAM. Namely, following the Ishai-Kushilevitz approach outlined above, we can plug the succinct garbling schemes for RAM into constant-round adaptively secure MPC (such as [DKR14, GP14]). The first protocol yields a fully adaptive MPC for RAM with 4 rounds; we refer to this protocol as "augmented [DKR14]".

The second construction, however, loses full security, since evaluating a garbling is a randomized functionality, and since their protocol doesn't guarantee secrecy of randomness of the function when everybody is corrupted. Namely, the simulator of the composed scheme will not be able to simulate the random coins of each party, since it needs to simulate generation randomness of the garbling scheme, consistent with simulated garbled values. This can be circumvented by using a garbling scheme where the simulator can also simulate random coins of the garbling, i.e. "adaptively secure" garbling [8] It is possible to construct such a garbling scheme by putting a mechanism allowing deniability (like in deniable encryption of [SW14]) on top of a garbling algorithm of RAM-efficient garbling scheme, say, [CH16], and obfuscating the whole circuit. This obfuscated circuit is a CRS of an adaptive garbling scheme[9]. Such a construction seems to give a RAM-efficient MPC protocol, which even allows to compute randomness-hiding functionalities (roughly, because the deniability mechanism of [SW14] generates random coins which are hidden from everybody). Still, this approach, which we call "augmented [GP14]", requires subexponentially-secure iO, and, since they use NIZK even in the semi-honest case, a local CRS.

In the table below we compare our result with existing work on constant round fully adaptive MPC([DKR14, GP14]), as well as with augmented versions of these protocols described above.

	Rounds	supports RAM	global CRS	supports randomness-hiding f	assumptions
[DKR14]	4	-	+	+	iO+OWF
[GP14]	2	-	-	-	subexp. iO+OWF
augmented [DKR14]	4	+	+	+	iO+OWF
augmented [GP14]	2	+	-	+	subexp. iO+OWF
our result	2	+	+	+	iO+OWF

Succinct NIZK proofs. To the best of our knowledge, the only approach for building NIZK proof systems where the length of the proof is independent of a circuit is based on encrypting satisfying assignment via FHE and making the verifier homomorphically evaluate the SAT circuit. This includes the work of [Gen09], who proposed the approach, and [Gro11], who shows how to bring the size of the proof down from $|w| \cdot \mathsf{poly}(\lambda)$ to $|w| + \mathsf{poly}(\lambda)$ (where w is the witness and λ is a security parameter); thus, the question of communication complexity of NIZK is resolved. However, in both schemes the verifier needs to do the work proportional to the circuit complexity of the function. Up to now we didn't know any fully succinct NIZK proof system (i.e. NIZK where both communication complexity and work of both parties is smaller than the circuit size).

[8] Note that usually the term "adaptive security" in the context of garbling is used to denote a different property: that the adversary can choose new inputs and functions after seeing garbled values.

[9] With this approach the environment has to fix inputs *before* seeing the CRS, i.e. this garbling scheme is only selectively secure. However, this is good enough for the protocol of [GP14], since they anyway use complexity leveraging and subexponentially-secure iO.

1.3 Our techniques: semi-honest case

Our MPC protocol takes a different approach than either of [GP14, DKR14, CGP15]. We present and motivate the approach.

First attempt. A natural idea for building MPC protocols is to use an obfuscated program to emulate a trusted party. That is, the CRS contains an obfuscated program which collects all inputs, does the computation, and outputs the result.

More precisely, the CRS should contain an encryption program Enc, which takes an input x_i and outputs its encryption c_i, and a decryption/evaluation program Eval, which takes $c_1, \ldots, c_n$, decrypts them, computes $y = f(x_1, \ldots, x_n)$ and outputs y. The parties can compute $f(x_1, \ldots, x_n)$ by encrypting $c_i = \mathsf{Enc}(x_i)$, broadcasting c_i, and computing $y \leftarrow \mathsf{Eval}(c_1, \ldots, c_n)$. However, such a protocol is clearly insecure: each party (say, P_1) can compute many different $y' = f(x'_1, x_2, \ldots, x_n)$ for any desired x'_1 by generating $c'_1 = \mathsf{Enc}(x'_1)$ and running $\mathsf{Eval}(c'_1, c_2, \ldots, c_n)$.

A natural way to mitigate such an attack is to make parties commit to their input first, and only then exchange ciphertexts and do the computation. Therefore we now have two rounds: in the first round parties exchange their commitments a_i, and in the second round they exchange ciphertexts c_i. To make sure that no party can run Eval on a different input than the one he committed to, Eval should check that x_i in c_i is consistent with the commitment a_i in the previous round. To achieve this, we need to put into c_i not only x_i, but also a_i together with its opening. Note however that this still allows a curious party to generate a different c'_i encrypting a different x'_i and a different, but valid commitment a'_i to x'_i, and then run Eval; thus we have to include *all* first-round commitments $a_1, \ldots, a_n$ within each c_i (together with an opening for a_i), so that a curious party couldn't modify its own a_i without being noticed.

At this point the protocol looks like this:

1. **The CRS:** Programs Enc and Eval, a CRS for a commitment scheme μ_{bind}
2. **Round 1:** Each party broadcasts $a_i \leftarrow \mathsf{Commit}(x_i)$, and keeps decommitment information r_i;
3. **Round 2:** Each party broadcasts $c_i \leftarrow \mathsf{Enc}(x_i; r_i; a_1, \ldots, a_n)$
4. **Evaluation:** Each party computes $y \leftarrow \mathsf{Eval}(c_1, \ldots, c_n)$

Here Eval decrypts each c_i and performs two checks: first, it checks that the set of $(a_1, \ldots, a_n)$ is the same in each c_i. Second, it checks that for all i r_i is a correct opening of a_i to x_i. If all checks pass, it outputs $f(x_1, \ldots, x_n)$.

While this idea works in general, the exact implementation becomes a challenge. Our goal is to show that a real execution is indistinguishable from a simulated one, where the simulated execution (and in particular, programs and communication) is generated by a simulator who doesn't know inputs of parties. One difficulty is to be able to switch the ciphertext from real (encrypting x_i) to simulated, and at the same time be able to generate Eval with the secret key of encryption inside. Several ways to accomplish this are known. One approach is to use a "double encryption + NIZK" paradigm ([NY90]); this method is chosen by [GP14] and it leads to a protocol secure against byzantine adversaries. However, one disadvantage of this approach is that the CRS is necessarily local.

The approach we take in order to switch c_i from real to simulated in the presence of the secret key is the "punctured key" technique, which guarantees that real and dummy ciphertexts are indistinguishable, even in the presence of "almost all" key - i.e. the key which decrypts everything except for this ciphertext. This allows us to first indistinguishably modify Eval such that it needs only a punctured key, and then switch a ciphertext (which the punctured secret key cannot decrypt) to a dummy ciphertext.

However, this approach has two shortcomings, which are not obvious from this discussion, but which would appear if we went deeper into the simulation and proofs. First, the technique requires hardwiring input-dependent values (such as x_i and c_i) into the program in the proof. This means that the inputs have to be fixed *before* the adversary sees Eval (and therefore the whole CRS), giving only *selective* security. Second, with this approach the programs in the simulated CRS have to contain simulated ciphertexts, and therefore we can only hope to get *local*, or *programmable*, CRS.

Second attempt. To solve both issues, we generate Enc and Eval during the runtime instead of fixing them in the CRS. Note that Enc is needed only in round 2 (and Eval is needed even later). Therefore we can let parties agree on generation randomness r_{Gen} in round 1, and then, after round 1 is complete, each party can run a special generation program Gen (which is now in the CRS instead of Enc and Eval) to produce a fresh pair of Enc and Eval, which are then used as before. In addition, we add to the CRS a special program $\mathsf{Explain}$, which inverts Gen, i.e. for any given output it produces consistent randomness r_{Gen}; this is used by the simulator only.

Therefore the protocol now looks like this:

- **The global CRS:** programs $\mathsf{Gen}, \mathsf{Explain}$, a CRS for a commitment scheme μ_{bind}
- **Round 1**: parties broadcast commitments $a_i = \mathsf{Commit}(x_i; r_i)$ together with randomness $r_{\mathsf{Gen},i}$;
- **After round 1:** each party sets generation randomness $r_{\mathsf{Gen}} \leftarrow \bigoplus r_{\mathsf{Gen},i}$ and obtains $\mathsf{Enc}, \mathsf{Eval} \leftarrow \mathsf{Gen}(r_{\mathsf{Gen}})$;
- **Round 2:** each party broadcasts $c_i \leftarrow \mathsf{Enc}(x_i; r_i; a_1, \ldots, a_n)$;
- **Evaluation:** each party computes $y \leftarrow \mathsf{Eval}(c_1, \ldots, c_n)$.

The simulator works as follows. First it generates programs $\mathsf{Enc}', \mathsf{Eval}'$ (which, as we said earlier, are different from real world programs). Next it uses $\mathsf{Explain}$ to generate randomness r_{Gen} on which Gen outputs these simulated $\mathsf{Enc}', \mathsf{Eval}'$. It generates all $r_{\mathsf{Gen},i}$ such that they xor to r_{Gen}, and sets a_i and c_i to be a dummy commitment and a dummy ciphertext. $(r_{\mathsf{Gen},i}, a_i, c_i)$ constitute simulated communications. To handle corruption of a party, the simulator equivocates the commitment; also the simulator needs to show the randomness for encryption, which it can do as long as underlying encryption is non-committing or deniable.

So far this scheme is only secure for a single execution. The problem is in the CRS of a statistically sound commitment scheme. Indeed, in the proof when we switch real world program Eval to simulated Eval' we crucially use the fact that commitments a_i are statistically binding to their inputs x_i. However, once the first execution is changed to the simulated, commitment CRS μ_{bind} has to be changed to the CRS μ_{equiv} which allows equivocation and thus is not binding anymore. Thus we cannot switch Eval from the second execution to simulated. (The other option is to switch executions from real to simulated in parallel, rather then sequentially; but this would require hardwiring variables for all executions at the same time into programs in the CRS, and thus the size of the CRS would have to grow with the number of executions.)

The other problem with a commitment scheme is that its CRS has to be local.

Third attempt. While this problem with commitments seems inevitable - one cannot hope to have a CRS which makes the commitment statistically binding and at the same time allows equivocation - what saves us is that commitment should be statistically binding *only at honestly generated commitments*, not everywhere. We call this primitive honest-but-curious (HBC) commitments.

Such a primitive can be easily constructed from one way functions: consider a length-doubling prg mapping $\{0,1\}^l$ to $\{0,1\}^{2l}$. For random $s \in \{0,1\}^l, r \in \{0,1\}^{2l}$, let $(\mathsf{prg}(s), r)$ be a commitment to 0 and $(r, \mathsf{prg}(s))$ be a commitment to 1. To open the commitment, show s. As long as a commitment was generated honestly, i.e. r was truly random, it doesn't have a valid prg preimage and therefore this commitment is statistically binding. The simulator can simulate the commitment by generating $\mathsf{prg}(s_0), \mathsf{prg}(s_1)$ and later open it to any bit. (Note that dishonest sender could cheat in the same way, and therefore binding holds only for honestly generated commitments. But it suffices for our MPC protocol, since we need a statistical binding property only for round 1 commitments a_i, which are generated by honest parties.)

Note that HBC commitments don't require a CRS, and therefore the CRS of the overall scheme is now global.

The choice of encryption scheme for the MPC protocol. As we said earlier, perhaps the most challenging part of the proof is to switch ciphertexts from real to simulated, while keeping the decryption key inside Eval. For this we take a punctured programming approach, and therefore we need an encryption scheme where it is possible to give a partial key, called a punctured key, which doesn't reveal anything about the challenge ciphertext. Our goal is the following: first we want to modify Eval so that it uses a punctured key instead of a real one; this should be done without changing the functionality of Eval, since we want to base security on iO. Importantly, modified Eval should not contain x_i, or any input-dependent values, since Eval should be generated by a simulator during the protocol execution, when the simulator might not know inputs of the parties yet. Next we want to use security of the punctured key and switch the ciphertext from real to simulated.

The puncturable deterministic encryption ([Wat15]), which is commonly used in this scenario, doesn't help us since the punctured program depends on inputs, making the simulation impossible. We therefore use a different encryption scheme, which we call a puncturable randomized encryption (PRE)[10]. In addition, this primitive may be viewed as a simulation-secure variant of PDE, and might be of independent interest.

Puncturable randomized encryption (PRE). In a definition of a semantically secure encryption scheme a real ciphertext is indistinguishable from a simulated one, even in the presence of a public key. A much stronger CCA security requires that ciphertexts are still indistinguishable even given access to a decryption oracle, i.e. to the functionality of a secret key everywhere except the challenge ciphertext. One can consider an ultimate version of CCA security and require that ciphertexts are indistinguishable even when *the secret key itself* is given in the clear (of course, for this to be meaningful, the secret key shouldn't be able to decrypt the challenge ciphertext, just like in case of standard definition of CCA-security). This is exactly what our puncturable randomized encryption achieves. In other words, a PRE scheme is secure under simulation

[10]Note that merely randomizing the PDE plaintext doesn't yield a PRE.

security definition, where the simulator needs to simulate a punctured key as well: that is, we require that a real-world punctured key and a ciphertext $(k\{c\}, c)$ are indistinguishable from simulated $(k\{c\}, c)$.

We build a secret key version of this primitive using puncturable PRFs and an injective public key encryption scheme (injective means that there doesn't exist a tuple (x, r, x', r') such that $(x, r) \neq (x', r')$ and $\mathsf{Enc}_{pk}(x; r) = \mathsf{Enc}_{pk}(x'; r')$). The secret key of a PRE consists of a public key of encryption scheme pk and a PRF key k. To encrypt a message m with randomness r, compute $T \leftarrow \mathsf{Enc}_{pk}(m; r)$, $C \leftarrow F_k(T) \oplus (m, r)$, and set the ciphertext to be (T, C). To decrypt (T, C), compute $(m, r) \leftarrow C \oplus F_k(T)$ and verify that $T = \mathsf{Enc}_{pk}(m; r)$.

To puncture a key at a ciphertext $(T^*, C^*) = \mathsf{PRE.Enc}(m; r)$, output $(pk, k\{T^*\})$, i.e. puncture PRF key k at T^*. This punctured PRE key doesn't give any information about plaintext of the ciphertext (T^*, C^*): intuitively, C^* looks uniformly random since k is punctured at T^*, and T^* itself doesn't reveal m since it is a ciphertext of a public key encryption. On the other hand, the punctured key still allows to encrypt all other plaintexts-randomness pairs and decrypt all other ciphertexts: note that for a given T there is only a single C which makes (T, C) a valid encryption; therefore puncturing out $k\{T^*\}$ affects exactly one valid ciphertext, i.e. (T^*, C^*).

The simulator can generate a dummy ciphertext (T^*, C^*) by setting $T^* \leftarrow \mathsf{Enc}_{pk}(0; r)$ and choosing C^* at random. It can also generate a corresponding punctured key as $(pk, k\{T^*\})$. This simulated ciphertext and punctured key $(T^*, C^*), (pk, k\{T^*\})$ can be shown to be indistinguishable from real ones by invoking security of a punctured PRF and an encryption scheme.

Computing randomness-hiding functionalities. Our protocol supports randomness-hiding functionalities. Essentially program Eval performs decryption of ciphertexts, computes a PRF on round-1 commitments and evaluates a randomized functionality which uses a PRF value as randomness. Intuitively, security of a PRF (and obfuscation on top of it) guarantees that this value remains hidden. The simulator can generate simulated Eval where this PRF is punctured and the result of the computation is hardcoded. For this idea to work it is important that Eval is generated during the runtime; if it was fixed in the CRS, we would have to hardwire outputs for every execution and therefore the CRS would have to grow with the number of executions.

Note that a perhaps more natural idea of randomizing the program Eval by applying a PRF to the inputs of the program (as it is, for instance, in probabilistic iO), instead of round-1 commitments, doesn't work here. Intuitively, this is because our approach for building this protocol crucially relies on the fact that round-1 messages completely determine the computation. This remains to be the case if randomness is computed as a PRF of round-1 messages, as we do in our protocol. But it doesn't hold anymore if inputs to Eval (i.e. round-2 ciphertexts) are used as an PRF input, since different round-2 ciphertexts may correspond to the same round-1 messages.

Achieving RAM efficiency. There are two ways to use our construction in order to achieve an efficient protocol. One way is to use iO for RAM in all programs involved. However, iO for RAM requires sub-exponential security of underlying iO for circuits. The other way, which only needs polynomially-secure iO for circuits, is to use the protocol to evaluate a functionality which takes parties' inputs and a function and outputs garbled function and garbled inputs; then parties can evaluate garbling themselves

locally. If a RAM-efficient garbling scheme is used ([CH16]), then the whole protocol becomes RAM-efficient. Note that it is enough to use *statically secure* garbling scheme, since our base protocol supports randomness-hiding functionalities, i.e. doesn't reveal randomness of the computation even when everybody is corrupted[11]. The composed scheme also supports randomized randomness-hiding functionalities: to evaluate such a functionality $f(x_1, \ldots, x_n; r)$, parties should use basic protocol to evaluate a randomized function $F(x_1, \ldots, x_n; (r_1, r_2))$ which uses r_1 as randomness to garble function f and inputs $x_1, \ldots, x_n, r_2$ (r_2 being random input of f).

1.4 Our techniques: malicious case

To obtain RAM efficiency in a malicious setting, we augment the protocol of [GP14]. Very roughly, in their protocol parties exchange commitments in round 1, and in round 2 they broadcast their input encrypted twice together with a NIZK proof that plaintexts are the same (the actual statement for the proof is more complicated, as discussed below). The CRS contains an obfuscated program which expects to see commitments from round 1, together with ciphertexts from round 2. This program checks NIZKs and uses a hardwired decryption key of a double encryption to decrypt the ciphertexts and evaluate the function. Each party can feed its transcript to this program and obtain the output.

So far the protocol seems to work in any model of computation: indeed, if we use iO for RAM to obfuscate the evaluation program in the CRS, then the work of each party becomes proportional to RAM complexity of a function. However, the problem is that the NIZK statement is more complicated than described above: it also requires proving that $y = f(x_1, \ldots, x_n)$, which is needed for the security proof to go through. As usual for "iO + NIZK" techniques, the NIZK has to be statistically sound. For all known NIZKs, this means that the verifier (in our case, the obfuscated evaluation program) has to do work proportional to the circuit complexity of f, even if the program is obfuscated with iO for RAM.

Therefore to make this protocol RAM-efficient, we show how to build RAM-efficient statistically sound NIZK.

RAM-efficient statistically sound NIZK for NP. Let a language L be specified by a relation $R(x, w)$. We build a statistically sound NIZK where, roughly, the work of the prover and NIZK length depends on $|R|_{\mathsf{RAM}}$, and the work of the verifier depends on worst-case RAM complexity of R.

Our main idea is the following: to prove that $x^* \in L$, the prover should send to a verifier a garbled program $\mathsf{GProg}(R(x, w))$, a garbled input $\mathsf{GInp}(x^*, w^*)$, and a NIZK proof (for circuits) that the garbling was done correctly: i.e. that the prover followed the garbling algorithm, and that it garbled correct function R and input x. The verifier should accept the proof if the NIZK proof verifies, and if the evaluation of a garbled program on a garbled input results in 1.

However, there are two issues. First, since we assume that we only have a NIZK for circuits, we need to make sure that the statement which we prove (i.e. that garbling was done correctly) is independent of the

[11] If the protocol revealed randomness of the computation, then the garbling scheme would have to be adaptively secure , i.e. the simulator of the garbling scheme would have to first simulate it and then, once it learned inputs, provide consistent generation randomness of the garbling scheme (note that the term "adaptive security" is ambiguous: in the context of garbling it usually denotes a different property, saying that simulation is possible even if inputs or functions are chosen adaptively after seeing some garbled values. Here by adaptive security we mean that random coins can be generated by the simulator).

circuit complexity of R (in particular, we need a garbling scheme where the size of circuits which generate garbling, i.e the size of GInp, GProg, only depend on a size of RAM description of a program to be garbled).

Second, note that this scheme guarantees that the garbler follows the garbling instructions (because of the NIZK), but there is no way to guarantee that the prover uses truly random coins to garble. This might introduce problems. Consider a garbling scheme which is not perfectly correct: say, for some choice of parameters the garbled program always outputs 1, no matter what the underlying program does[12]. In this case a malicious and unbounded prover could choose these bad parameters and therefore convince the verifier in wrong statements, since the evaluation of a garbled program results in 1 no matter whether $R(x, w)$ holds or not. Thus, we need a garbling scheme where the evaluation can never result in the wrong answer, i.e. where the computation *always* results in either a correct result or $\perp$. We call this property *perfect correctness with abort*.

We note that the garbling scheme of Canetti and Holmgren ([CH16]) already has both properties; see appendix A for details. Thus, our scheme yeilds a NIZK system when instantiated with the garbling scheme by [CH16].

2 Building blocks

In this section we define and build a *puncturable randomized encryption (PRE)* and *an honest-but-curious commitment* - primitives used in our MPC protocol (section 3).

2.1 Puncturable randomized encryption

Puncturable randomized encryption (PRE) is a randomized, symmetric key encryption. Besides standard algorithms $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}$, there is additional procedure $\mathsf{Puncture}(k, c^*)$ which takes as input a key k and a ciphertext $c^* = \mathsf{Enc}(m^*; r^*)$ and outputs a partial, or *punctured*, key $k\{c^*\}$. Such a key has two properties. First, it doesn't reveal any information about the plaintext of c^*; this is captured by requiring that a simulator should simulate a ciphertext and a punctured key without knowing a plaintext. Second, the key should still have the same functionality in all other points: namely, it should correctly decrypt all other $c \neq c^*$, and it should correctly encrypt all other $(m, r) \neq (m^*, r^*)$.

PRE can be viewed as a randomized, simulation-secure analog of a puncturable deterministic encryption (PDE) [SW14].

Definition 1. *Puncturable randomized encryption (PRE) is a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Puncture}, \mathsf{Sim})$*, which satisfy the following properties:*

- **Statistical correctness:** *With overwhelming probability over the choice of the key* $k \leftarrow \mathsf{Gen}(1^\lambda)$*, for any message* m *and randomness* r $\mathsf{Dec}_k(\mathsf{Enc}_k(m; r)) = m$.
- **Statistical correctness of the punctured key:** *With overwhelming probability over the choice of the key* $k \leftarrow \mathsf{Gen}(1^\lambda)$*, for any message* m^* *and randomness* r^**, let* $c^* \leftarrow \mathsf{Enc}_k(m^*; r^*)$*, and* $k\{c^*\} \leftarrow \mathsf{Puncture}(k, c^*)$*. Then:*

[12] Note that the proof of garbling done correctly doesn't save us, since the garbler followed the garbling algorithm; it's just the scheme itself allows for wrong garbling.

– *for any* (m, r) *such that* $(m, r) \neq (m^*, r^*)$, $\mathsf{Enc}_k(m; r) = \mathsf{Enc}_{k\{c^*\}}(m; r)$*;*

– *for any* $c \neq c^*$ $\mathsf{Dec}_k(c) = \mathsf{Dec}_{k\{c^*\}}(c)$ *(in particular, both decryptions should output* $\perp$ *on the same set of ciphertexts, except* c^**).*

- **Simulation security with the punctured key:** *For any PPT adversary A and for any message* m^**, consider the following experiment:* $k \leftarrow \mathsf{Gen}(1^\lambda)$*,* r^* *is chosen at random,* $c^* \leftarrow \mathsf{Enc}_k(m^*; r^*)$*,* $k\{c^*\} \leftarrow \mathsf{Puncture}(k, c^*)$*, and* $(c_{\mathsf{Sim}}, k\{c_{\mathsf{Sim}}\}) \leftarrow \mathsf{Sim}()$*. Then*

$$\mathsf{Pr}[A(k\{c^*\}, m^*, c^*) = 1] - \mathsf{Pr}[A(k\{c_{\mathsf{Sim}}\}, m^*, c_{\mathsf{Sim}}) = 1] < \mathsf{negl}(\lambda).$$

Simulation security says that even if an adversary has almost all key, it cannot tell whether it sees an encryption of a known message m^* or a simulated encryption (as long as randomness of encryption remains hidden). Note that simulation security with the punctured key implies normal security of PRE as a secret-key encryption, since with $k\{c^*\}$ the adversary can answer encryption-decryption queries itself.

Our construction in a nutshell. The key of a PRE consists of a key K of a puncturable PRF and a public key pk of an injective encryption scheme. To encrypt message m under randomness r, the sender computes $T \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m; r)$, $C \leftarrow F_K(T) \oplus (m, r)$, and sets its ciphertext to be (T, C). To decrypt, the receiver computes $(m, r) \leftarrow F_K(T) \oplus C$ and checks whether $T = \mathsf{Enc}_{\mathsf{pk}}(m; r)$. To puncture the key at a ciphertext (T, C), output $(pk, K\{T\})$, where $K\{T\}$ is a PRF key punctured at T.

In this construction the encryption scheme should be injective for both message and randomness. We observe that the encryption scheme by [SW14], where the ciphertext is $(\mathsf{prg}(r), F_k(\mathsf{prg}(r)) \oplus m)$, satisfies this property, as long as the underlying prg is injective. In turn, (the family of) injective prgs exists assuming iO and injective OWFs: indeed, the fact that $\mathsf{iO}(\mathsf{PRF})$ is a hardcore function [BST14] immediately implies that this is also a prg family; this prg can be made injective by putting an injective PRF [SW14] inside.

Therefore we obtain PRE assuming iO and injective OWFs.

More detailed description. We construct PRE from puncturable PRFs and a public key encryption which is injective with respect to both message and randomness (i.e. it should hold that $\mathsf{Enc}_{\mathsf{pk}}(m_1; r_1) = \mathsf{Enc}_{\mathsf{pk}}(m_2; r_2)$ implies $(m_1, r_1) = (m_2, r_2)$).

Lemma 1. *[SW14, BST14] Assuming indistinguishability obfuscation for circuits and injective one way functions, there exists a public key encryption which is statistically injective with respect to both message and randomness.*

Proof. In short, the work of [BST14] essentially builds an injective prg, which can be plugged into encryption scheme of [SW14] to obtain injective PKE. We briefly present all constructions here for completeness.

Overall encryption scheme. Recall that in the PKE scheme of [SW14] the public key is an obfuscated program which takes (m, r) as input, computes $t = \mathsf{prg}(r)$, and outputs $(t, F_k(t) \oplus m)$ as a ciphertext. Note that this scheme is only injective for messages, but not for randomness, since underlying prg could map two different randomness to the same output. Thus for this encryption to be injective, we need an injective prg. In addition, note that for this construction it is enough to have *a family* of prgs (which is statistically injective): the prg could be chosen from the family during the process of the key generation for the encryption scheme.

Injective prg family. We note that the work of Bellare *et al.*[BST14], which proves that $\mathsf{iO}(\mathsf{PRF})$ is a hardcore function for any injective OWF[13], also implies that $\mathsf{iO}(\mathsf{PRF})$ is a prg family. Indeed, in their work they show that $H = \mathsf{iO}(\mathsf{PRF})$ is a hardcore function for any injective OWF f, i.e. that for random r $(f, H, f(r), H(r)) \approx_c (f, H, f(r), U_{|H(r)|})$. This implies the following: as long as there exists an injective OWF f, it holds that $(f, H, f(r), H(r)) \approx_c (f, H, f(r), U_{|H(r)|})$ and therefore it also holds that $(H, H(r)) \approx_c (H, U_{|H(r)|})$, which means that this is a prg family.

This prg family is statistically injective, as long as the underlying PRF is statistically injective.

Injective PRF family. Sahai and Waters [SW14] build a statistically injective puncturable PRF family from a PRF family $\{F_k(x)\}$ (which in turn can be built from OWFs) and a 2-universal hash function $h(x)$ (which exists unconditionally) as $F_k(x) \oplus h(x)$, as long as the output of a PRF is large enough. Namely, they show that as long as $m(\lambda) > 2n(\lambda) + e(\lambda)$, there exists such a statistically injective PRF family which maps $n(\lambda)$ bits to $m(\lambda)$ bits and has a failure probability $2^{-e(\lambda)}$ (i.e. with probability $2^{-e(\lambda)}$ over the choice of the PRF key the PRF is not injective).

This concludes the proof that a statistically injective PKE exists assuming iO and injective OWFs. We underline that this PKE is only statistically injective, since underlying PRFs might be non-injective with some negligible probability. □

From injective PKE to PRE. Our PRE is constructed as follows (see fig. 1 for a more concise description):

- **Key generation:** $\mathsf{PRE.Gen}(1^\lambda, r_{\mathsf{Gen}})$ uses r_{Gen} to sample a PRF key K and generate $(\mathsf{pk}, \mathsf{sk})$-pair of a public key encryption scheme which is statistically injective for messages and randomness. It sets $\mathsf{PRE}.k \leftarrow (K, \mathsf{pk})$.
- **Encryption:** $\mathsf{PRE.Enc}_{\mathsf{PRE}.k}(m; r)$ sets $T \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m; r)$ and $C \leftarrow F_K(T) \oplus (m, r)$ (if the key K is punctured at point T, encryption outputs $\perp$). It outputs the ciphertext $c = (T, C)$.
- **Decryption:** $\mathsf{PRE.Dec}_{\mathsf{PRE}.k}(c)$ parses c as (T, C) and sets $(m, r) \leftarrow F_K(T) \oplus C$ (if the key K is punctured at point T, decryption outputs $\perp$). Next it verifies that $\mathsf{Enc}_{\mathsf{pk}}(m; r) = T$; if this check passes, it outputs m, otherwise it outptus $\perp$.
- **Puncture:** $\mathsf{PRE.Puncture}(\mathsf{PRE}.k, c)$ parses c as (T, C) and punctures the PRF key at T; it outputs the PRE punctured key $(pk, K\{T\})$.
- **Simulation:** $\mathsf{PRE.Sim}()$ first chooses the key $\mathsf{PRE}.k$ by sampling a PRF key K and generating $(\mathsf{pk}, \mathsf{sk})$-pair of a public key encryption scheme. Next it generates $T = \mathsf{Enc}_{\mathsf{pk}}(0; r)$ for random r and sets C to be a random string. It sets the simulated ciphertext c_{Sim} to be (T, C) and outputs it. Next, it punctures the PRF key K at T and sets the simulated punctured key $k\{c_{\mathsf{Sim}}\}$ to be $(pk, K\{T\})$.

Theorem 4. *Assuming that PKE is a public key encryption scheme, injective for both messages and randomness, and assuming one way functions, the construction presented on fig. 1 is a puncturable randomized encryption.*

[13] In fact, for them it is enough that OWF is poly-to-one. Thus we can relax our assumptions for MPC protocol from injective OWF to poly-to-one OWF.

Construction of a PRE

$\mathsf{PRE.Gen}(1^\lambda, r_{\mathsf{Gen}})$:
1. Sample PRF.K and $(\mathsf{PKE.pk}, \mathsf{PKE.sk})$;
2. Output (PRF.K, PKE.pk)

$\mathsf{PRE.Enc}_{\mathsf{PRE}.k}(m; r)$:
1. $T \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m; r)$
2. If K is punctured at T, output $\perp$ and halt;
3. $C \leftarrow F_K(T) \oplus (m, r)$.
4. outputs (T, C).

$\mathsf{PRE.Dec}_{\mathsf{PRE}.k}(T, C)$:
1. If K is punctured at T, output $\perp$ and halt;
2. $(m, r) \leftarrow F_K(T) \oplus C$
3. If $\mathsf{Enc}_{\mathsf{pk}}(m; r) = T$ then output m, else $\perp$.

$\mathsf{PRE.Puncture}(\mathsf{PRE}.k, c = (T, C))$:
1. Output $\mathsf{PRE}.k\{c\} = (pk, K\{T\})$

$\mathsf{PRE.Sim}()$:
1. $\mathsf{PRE}.k \leftarrow \mathsf{PRE.Gen}(r_{\mathsf{Gen}})$ for random r_{Gen};
2. $T = \mathsf{Enc}_{\mathsf{pk}}(0; r)$ for random r;
3. $C \leftarrow$ random ;
4. output $c = (T, C)$, $\mathsf{PRE}.k\{c\} = (pk, K\{T\})$;

Figure 1: Construction of a PRE from a puncturable PRF and injective PKE.

Proof. Before showing correctness and security, we note the following useful property of our encryption:

First part of a ciphertext determines the second. For a given T^*, there exists at most one C^* such that (T^*, C^*) is a valid (i.e. decrypted to non-$\perp$) ciphertext. Indeed, due to injectivity of underlying PKE, there exists at most one (m^*, r^*) pair such that $T^* = \mathsf{PKE.Enc}_{\mathsf{pk}}(m^*; r^*)$. Therefore the check in the decryption algorithm will only pass for $C^* = F_K(T^*) \oplus (m^*, r^*)$.

Correctness. This scheme is statistically correct, as immediately follows from correctness of encryption $C = F_K(T) \oplus (m, r)$ and the fact that the check $T = \mathsf{Enc}_{\mathsf{pk}}(m; r)$ passes for honestly generated ciphertext.

Next, correctness of the punctured key also holds, as long as underlying PKE is injective: indeed, there is only a single (m, r)-pair which results in $T = T^*$, and therefore puncturing out T^* in k only affects encryption of m^* with r^*. On a decryption side, since only (T^*, C^*) is a valid ciphertext with $T = T^*$, puncturing k only affects the decryption of (T^*, C^*). Indeed, ciphertexts of the form $(T \neq T^*, C)$ are decrypted in the same way regardless of which key is used, the full key or the punctured one. On the other hand, ciphertexts of the form $(T^*, C \neq C^*)$ are rejected by decryption with both real and punctured keys: indeed, decryption with the full key rejects it since the ciphertext is invalid, and decryption with the punctured key rejects it since decryption tries to evaluate the PRF at the punctured point T^*, so the check in

line 1 of decryption fails.

Security. To show security, we need to show that the punctured key, the message, and the ciphertext, i.e. $((K\{T^*\}, pk), m^*, (T^*, C^*))$, is indistinguishable in the two cases: in one case $T^* = \mathsf{Enc}_{\mathsf{pk}}(m^*; r^*)$, $C^* = F_K(T^*) \oplus (m^*, r^*)$, and in the other case $T^* = \mathsf{Enc}_{\mathsf{pk}}(0)$ and C^* is randomly chosen. We do this by considering a middle distribution where T^* is real, i.e. $T^* = \mathsf{Enc}_{\mathsf{pk}}(m^*; r^*)$, but C^* is random. The middle and the real distribution are indistinguishable due to the property of a punctured PRF: $F_K(T^*)$ is indistinguishable from random, therefore so is $F_K(T^*) \oplus (m^*, r^*)$. Middle and simulated distributions are indistinguishable by security of a PKE. □

2.2 Honest-but-curious Equivocal Commitments

Motivated by the fact that standard non-interactive commitments are unnecessary strong for our protocol (i.e. support malicious behavior of the sender) and at the same time make the CRS local, we consider a weaker semi-honest commitment which doesn't have this disadvantage.

Namely, an honest-but-curious commitment scheme $(\mathsf{HBCCommit}, \mathsf{Verify})$ can be used to commit to a value x with randomness r using $c \leftarrow \mathsf{HBCCommit}(x; r)$, which later can be opened to convince the verifier that it was x that was committed to. The difference between this primitive and the standard commitment is in the security guarantee. Here we only require that an *honestly* generated commitment cannot be opened in a different way, even by an unbounded adversary. The other way to state this property is to say that for overwhelming fraction of randomness, commitments are statistically binding; this means that a semi-honest sender will generate a statistically binding commitment. (Still, there can be a negligible fraction of commitments which can be easily opened in both ways).

In addition, we require the commitment scheme to be equivocal, or adaptively secure, i.e. the simulator should be able to provide randomness consistent with the simulated commitment.

Unlike its stronger counterpart, honest-but-curious commitment can be constructed in a plain model, in a fairly simple way.

Definition 2. *An honest-but-curious commitment scheme for a message space M is a pair of PPT algorithms $(\mathsf{HBCCommit}(x; r), \mathsf{Verify}(x, r, c))$, such that the following properties hold:*

- **Correctness:** *For any x, r $\mathsf{Verify}(x, r, \mathsf{HBCCommit}(x; r)) = 1$;*
- **Most commitments are statistically binding:** *For any $x \in M$*

$$\Pr_r[\exists r', x' \text{ s.t. } x' \neq x \wedge \mathsf{Verify}(x', r', \mathsf{HBCCommit}(x; r)) = 1] < \mathsf{negl}(\lambda).$$

- **Computational hiding and equivocation:** *There exist a PPT simulator Sim such that for any $x \in M$ it holds that*

$$\{(r, x, c) : c \leftarrow \mathsf{HBCCommit}(x; r), r \leftarrow \{0,1\}^{|r|}\} \approx_c$$
$$\{(r, x, c) : (c, \mathsf{state}) \leftarrow \mathsf{Sim}(), r \leftarrow \mathsf{Sim}(x, \mathsf{state})\}.$$

Construction. We build a semi-honest commitment scheme for message space $M = \{0, 1\}$. Consider a prg with exponentially sparse range (say, length-doubling prg, mapping λ bits to 2λ bits). To commit to 0, output $(\mathsf{prg}(s), r)$, and to commit to 1, output $(r, \mathsf{prg}(s))$, where s is a random value of size λ, and r is a random value of size 2λ. To open the commitment, show (s, r).

Since honestly generated (i.e. random) r is outside the image of the prg with overwhelming probability, there is no s such that $\mathsf{prg}(s) = r$, and therefore for honestly generated commitment there doesn't exist the wrong opening. On the other hand, the simulator can generate its commitment as $(\mathsf{prg}(s_0), \mathsf{prg}(s_1))$ and later open it to any bit b, showing s_b and claiming that the other value is randomly chosen. Thus we proved the following statement:

Theorem 5. *Assuming the existence of one way functions, the above scheme is an honest-but-curious commitment scheme for the message space $M = \{0, 1\}$.*

3 Our MPC protocol against semi-honest adversaries

In this section we present our two-round, RAM-efficient, semi-honest protocol with global CRS.

Our protocol is described in Fig. 2 and corresponding programs are given in Fig. 3, Fig. 4. The CRS consists of two programs, Gen and ExplainGen. Gen is a generation algorithm which produces "encryption" program B, "decryption-and-evaluation" program Eval and program ExplainB. Both ExplainGen and ExplainB are not used in the protocol execution; they are used in the simulation only in order to provide consistent randomness for Gen and B.

In the first round everybody uses the semi-honest commitment scheme (defined and constructed in section 2.2) to "commit" to (i, x_i) with randomness $r_{\mathsf{com},i}$. In addition, parties exchange randomness $r_{\mathsf{Gen},i}$ and everybody sets (the same) $r_{\mathsf{Gen}} \leftarrow \bigoplus r_{\mathsf{Gen},i}$. Everybody runs $\mathsf{Gen}(r_{\mathsf{Gen}})$ to obtain the same programs $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$.

In round 2 everybody runs $b_i \leftarrow \mathsf{B}(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n; r_{B,i})$ (which essentially encrypts all round 1 messages together with a party's own opening of a commitment, under some randomness $r_{B,i}$) and sends out b_i. Then everybody computes $y \leftarrow \mathsf{Eval}(b_1, \ldots, b_n)$. Eval decrypts every ciphertext, validates each commitment using opening provided in corresponding ciphertext, and in addition checks that all ciphertexts agree on the set of round-one commitments. If these checks pass, Eval does the computation (computing randomness as a PRF of commitments, if the function is randomized) and outputs y.

The protocol

CRS: programs Gen and ExplainGen
inputs: x_i; randomness: $r_{\mathsf{com},i}, r_{B,i}, r_{\mathsf{Gen},i}$

1. **Round 1:** Each party P_i computes $a_i \leftarrow \mathsf{HBCCommit}(i, x_i; r_{\mathsf{com},i})$ and broadcasts $(a_i, r_{\mathsf{Gen},i})$;
2. Each party sets $r_{\mathsf{Gen}} \leftarrow \bigoplus r_{\mathsf{Gen},i}$ and runs $\{\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}\} \leftarrow \mathsf{Gen}(r_{\mathsf{Gen}})$;
3. **Round 2:** Each party broadcasts $b_i \leftarrow \mathsf{B}(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n; r_{B,i})$;
4. Each party sets its output to be $y \leftarrow \mathsf{Eval}(b_1, \ldots, b_n)$.

Figure 2: MPC protocol.

The central encryption scheme used by program B to encrypt and by Eval to decrypt is a puncturable randomized encryption (PRE), which we built in section 2.1) from iO and injective OWFs. In addition, both Gen and B have a trapdoor branch which helps the simulator to generate consistent randomness with the help of programs $\mathsf{ExplainGen}, \mathsf{ExplainB}$. Essentially helper programs $\mathsf{ExplainGen}, \mathsf{ExplainB}$ use a special encryption scheme (puncturable deterministic encryption, PDE, [Wat15]), in order to encode an instruction "output $output^*$ and halt" into a random-looking value, which pretends to be true randomness of a party. Gen and B try to decrypt this value in a trapdoor branch and follow the instruction encoded. In addition, this technique requires to use a special PRF, called extracting PRF, F_{Ext} ([SW14]) We don't elaborate on this mechanism further since it closely follows the original idea of [SW14], [DKR14].

Theorem 6. *Assuming injective one way functions*[14] *and indistinguishability obfuscation for circuits, the presented protocol is a two-round multiparty protocol with global CRS adaptively secure against honest-but-curious corruptions of possibly all parties. The protocol allows to compute any randomized functionalities, even randomness-hiding ones. Its communication complexity depends on* $\lambda, \{|x_i|\}_{i=1}^n, y, |f|_{\mathsf{RAM}}$ *(logarithmic parameters omitted), and time and space of every party depends on* $\lambda, \{|x_i|\}_{i=1}^n, y, |f|_{\mathsf{RAM}}$*, and time or space needed to evaluate RAM* $f(x_1, \ldots, x_n)$ *in the worst case.*

Programs in the CRS:

Program $\mathsf{Gen}(r_{\mathsf{Gen}})$
Constants: an extracting PRF key $\mathsf{Ext}_{\mathsf{Gen}}$, faking PDE key f_{Gen}

- **Trapdoor branch:**
  1. set $(\mathsf{Prog1}, \mathsf{Prog2}, \mathsf{Prog3}, \tilde{\rho}) \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{Gen}}}(r_{\mathsf{Gen}})$. If decryption returns $\perp$ then goto normal branch;
  2. output $\mathsf{Prog1}, \mathsf{Prog2}, \mathsf{Prog3}$ and halt;
- **Normal branch:**
  1. $u_{\mathsf{Gen}} \leftarrow F_{\mathsf{Ext}_{\mathsf{Gen}}}(r_{\mathsf{Gen}})$;
  2. use u_{Gen} to sample extracting PRF key $\mathsf{Ext}_{\mathsf{B}}$, PRE key K, PRF key k, faking PDE key f_{B} and obfuscation randomness for $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$;
  3. output obfuscated programs $\mathsf{B}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K], \mathsf{Eval}[K, k], \mathsf{ExplainB}[f_{\mathsf{B}}]$.

Program $\mathsf{ExplainGen}(\mathsf{Prog1}, \mathsf{Prog2}, \mathsf{Prog3}; \rho)$
Constants: faking PDE key f_{Gen}

1. Set $M = ((\mathsf{Prog1}, \mathsf{Prog2}, \mathsf{Prog3}), \mathsf{prg}(\rho))$;
2. Set $r_{\mathsf{Gen}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Gen}}}(M)$;
3. output r_{Gen}.

Figure 3: Programs in the CRS of our protocol. Program Gen chooses keys and outputs obfuscated programs $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$, defined in figure 4. Program $\mathsf{ExplainGen}$ is only used by the simulator in order to generate consistent random coins for Gen.

[14] In fact, this requirement can be relaxed down to one way functions with at most polynomial-size preimage, since such OWF suffices to prove that the construction of [BST14] is secure; and therefore the PRE scheme (section 2.1) exists under this assumption and iO.

Programs produced by the CRS:

Program $\mathsf{B}(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n; r_{B,i})$
Constants: an extracting PRF key Ext_B, faking PDE key f_B, PRE key K

- **Trapdoor branch:**
  1. set $(i', x', r'_{\mathsf{com},i}, a'_1, \ldots, a'_n, b', \tilde{\rho}) \leftarrow \mathsf{PDE.Dec}_{f_\mathsf{B}}(r_{B,i})$. If decryption returns $\perp$ then goto normal branch;
  2. if $(i', x', r'_{\mathsf{com},i}, a'_1, \ldots, a'_n) \neq (i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$ then goto normal branch;
  3. output b' and halt;
- **Normal branch:**
  1. Set $M = (i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
  2. $u_{B,i} \leftarrow F_{\mathsf{Ext}_\mathsf{B}}(M, r_{B,i})$
  3. Set $b \leftarrow \mathsf{PRE.Enc}_K(M; \mathsf{prg}(u_{B,i}))$
  4. Output b

Program $\mathsf{Eval}(b_1, \ldots, b_n)$
Constants: PRE key K, key k of a PRF G

1. For every i decrypt:
   (a) Set $M_i \leftarrow \mathsf{PRE.Dec}_K(b_i)$;
   (b) Parse M_i as $(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$. If the format is wrong (in particular, if i is wrong), output $\perp$.
2. For every i check consistency:
   (a) Verify that the set $(a_1, \ldots, a_n)$ is the same in all $M_1, \ldots, M_n$;
   (b) Verify that $a_i = \mathsf{HBCCommit}(i, x_i; r_{\mathsf{com},i})$
3. Set $R \leftarrow G_k(a_1, \ldots, a_n)$.
4. Output $y \leftarrow f(x_1, \ldots, x_n; R)$. (If f is deterministic, ignore R).

Program $\mathsf{ExplainB}(i, x, r_{\mathsf{com},i}, a_1, \ldots, a_n; b; \rho)$
Constants: PDE key f_B

1. Set $M = ((i, x, r_{\mathsf{com},i}, a_1, \ldots, a_n), b, \mathsf{prg}(\rho))$
2. Set $r_{B,i} \leftarrow \mathsf{PDE.Enc}_{f_\mathsf{B}}(M)$
3. output $r_{B,i}$

Figure 4: Programs used in the protocol.

4 The Proof of the Main Theorem

On achieving RAM efficiency. There are two ways to use our construction in order to achieve an efficient protocol. One way is to use iO for RAM in all programs involved (in particular, the program Gen, which obfuscates three programs, should use an obfuscator for RAM). The other way is to use the protocol to evaluate a functionality which takes parties' inputs and a function and outputs garbled function and garbled inputs; then parties can evaluate garbling themselves locally. If a RAM-efficient garbling scheme is used ([CH16]), then it suffices to use iO for circuits to make the whole protocol RAM-efficient. Note that it is enough to use statically secure garbling scheme, since our base protocol supports randomness-hiding func-

tionalities, i.e. doesn't reveal randomness of the computation even when everybody is corrupted[15]. The composed scheme also supports randomized randomness-hiding functionalities: to evaluate such a functionality $f(x_1, \ldots, x_n; r)$, parties should use basic protocol to evaluate a randomized function $F(x_1, \ldots, x_n; (r_1, r_2))$ which uses r_1 as randomness to garble function f and inputs $x_1, \ldots, x_n, r_2$ (r_2 being random part of input).

Unlike the first approach, the second approach doesn't require subexponentially-secure iO (which is an assumption currently required for iO for RAM).

In both cases, we assume that the simulator gets all necessary information about the computation (such as worst-case running time, space, etc) from the ideal functionality. As discussed in the introduction, setting a lower (than the worst-case) bound on the running time/space of the computation might be useful if parties agree to sacrifice some security for efficiency.

Correctness. Correctness of the scheme can be immediately verified. Note that in case of randomized functionalities the randomness for the computation is obtained via a PRF G, and therefore the distribution of the output is only computationally close to the ideal distribution.

Simulation. The simulator works as follows:

CRS: The simulator generates the CRS honestly.

Round 1: Each a_i^* is simulated by a simulator of a semi-honest commitment scheme. Each b_i^* is simulated by $\mathsf{PRE.Sim}$, together with a punctured key $K\{\{b_i^*\}_{i=1}^n\}$. $\mathsf{Eval1}, \mathsf{B1}$ are generated as in fig. 5 (using punctured keys $K\{\{b_i^*\}_{i=1}^n\}$ and $k\{(a_1^*, \ldots, a_n^*)\}$), and $\mathsf{ExplainB}$ is generated as in fig. 3. r^*_{Gen} is set to explain these $\mathsf{B1}, \mathsf{Eval1}, \mathsf{ExplainB}$ (i.e. it is generated as $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}(\mathsf{Eval1}, \mathsf{B1}, \mathsf{ExplainB}; \rho)$ for random ρ). Each $r^*_{\mathsf{Gen},i}$ is set to sum up to r^*_{Gen}. $(a_i^*, r^*_{\mathsf{Gen},i})$ is a simulated first message of each party.

Round 2: b_i^* (generated in round 1) is a simulated second message of each party.

Simulating internal state: $r^*_{\mathsf{com},i} \leftarrow \mathsf{HBCCommit.Sim}(a_i^*, x_i)$ is generated, and $r^*_{B,i}$ is set to explain b_i^* on input $(i, x_i^*, r^*_{\mathsf{com},i}, a_1^*, \ldots, a_n^*)$ (i.e. it is generated as $r^*_{B,i} \leftarrow \mathsf{ExplainB}((i, x_i^*, r^*_{\mathsf{com},i}, a_1^*, \ldots, a_n^*), b_i^*; \rho_i)$) for some random ρ_i. $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is internal state of each party.

Simulator's knowledge of the output. Note that the simulator is required to hardwire the output y^* into $\mathsf{Eval1}$ (fig. 5); $\mathsf{Eval1}$ has to be generated at the end of round 1, since r^*_{Gen} (which is determined right after round 1 ends) depends on it. It could be that at that moment nobody is corrupted, and the simulator, formally speaking, doesn't know the output y^*.

However, we can always assume that it knows y^* as soon as the simulation starts. The idea is similar to the idea allowing parties to compute different outputs: they should evaluate a different function $f'((x_1, r_1), \ldots, (x_n, r_n)) = f_1(x_1, \ldots, x_n) \oplus r_1 || \ldots || f_n(x_1, \ldots, x_n) \oplus r_n$, where r_i is randomness chosen by party i. In this new protocol the simulator can set the output to be a random value z (which can be

[15] If the protocol revealed randomness of the computation, then the garbling scheme would have to be adaptively secure , i.e. the simulator of the garbling scheme would have to first simulate it and then, once it learned inputs, provide consistent generation randomness of the garbling scheme (note that the term "adaptive security" is ambiguous: in the context of garbling it usually denotes a different property, saying that simulation is possible even if inputs or functions are chosen adaptively after seeing some garbled values. Here by adaptive security we mean that random coins can be generated by the simulator).

chosen even before the protocol starts), and as soon as party i is corrupted and the simulator learns y_i, it can set $r_i \leftarrow z_i \oplus y_i$ (where z_i is the i-th block of z corresponding to the output of party i).

Programs used in the proof and the simulation

Program $\mathsf{B1}(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n; r_{B,i})$

Constants: an extracting PRF key Ext_B, faking PDE key f_B, punctured PRE key $K\{\{b_i^*\}_{i=1}^n\}$

- **Trapdoor branch:**
  1. set $(i', x', r'_{\mathsf{com},i}, a'_1, \ldots, a'_n, b', \tilde{\rho}) \leftarrow \mathsf{PDE.Dec}_{f_\mathsf{B}}(r_{B,i})$. If decryption returns $\perp$ then goto normal branch;
  2. if $(i', x', r'_{\mathsf{com},i}, a'_1, \ldots, a'_n) \neq (i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$ then goto normal branch;
  3. output b' and halt;
- **Normal branch:**
  1. Set $M = (i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
  2. $u_{B,i} \leftarrow F_{\mathsf{Ext}_\mathsf{B}}(M, r_{B,i})$
  3. Set $b \leftarrow \mathsf{PRE.Enc}_{K\{\{b_i^*\}_{i=1}^n\}}(M; \mathsf{prg}(u_{B,i}))$
  4. Output b

Program $\mathsf{Eval1}(b_1, \ldots, b_n)$

Constants: punctured PRE key $K\{\{b_i^*\}_{i=1}^n\}$, punctured PRF key $k\{(a_1^*, \ldots, a_n^*)\}$, $a_1^*, \ldots, a_n^*$, $b_1^*, \ldots, b_n^*, y^*$

Case 0: If there is $i \neq j$ such that $b_i = b_j^*$, output $\perp$.

Case 1: If for all i $b_i = b_i^*$, then output y^* and halt.

Case 2: If for some i $b_i = b_i^*$ (denote such set as $\mathcal{I}$), then:

1. For every $i \notin \mathcal{I}$ decrypt:
   (a) Set $M_i \leftarrow \mathsf{PRE.Dec}_{K\{\{b_i^*\}_{i=1}^n\}}(b_i)$;
   (b) Parse M_i as $(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
2. For every $i \notin I$ check consistency:
   (a) Verify that the set $(a_1, \ldots, a_n)$ is the same as $(a_1^*, \ldots, a_n^*)$
   (b) Verify that $a_i = \mathsf{HBCCommit}(i, x_i; r_{\mathsf{com},i})$
3. Output y^*.

Case 3: If for all i $b_i \neq b_i^*$, then:

1. For every i decrypt:
   (a) Set $M_i \leftarrow \mathsf{PRE.Dec}_{K\{\{b_i^*\}_{i=1}^n\}}(b_i)$;
   (b) Parse M_i as $(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
2. For every i check consistency:
   (a) Verify that the set $(a_1, \ldots, a_n)$ is the same in all $M_1, \ldots, M_n$;
   (b) Verify that $a_i = \mathsf{HBCCommit}(i, x_i; r_{\mathsf{com},i})$
3. If $(a_1, \ldots, a_n) = (a_1^*, \ldots, a_n^*)$ then output y^*
4. Set $R \leftarrow G_{k\{(a_1^*, \ldots, a_n^*)\}}(a_1, \ldots, a_n)$.
5. Output $y \leftarrow f(x_1, \ldots, x_n; R)$.

Figure 5: Programs used in the proof and the simulation.

Leakage Resilience. For an adaptively secure protocol to be leakage resilient, the simulator has to be *corruption oblivious*, i.e. when simulating leakage from a party, the simulator can only use ideal-world leakage from *this party*; even if some information was leaked from other parties before (and therefore the simulator knows the information and simulated leakage), it cannot be used in simulation of leakage of the current party.

A convenient way to think about this is to imagine that the simulator S should have special subroutines $S_1, \ldots, S_n$ (each S_i handles leakage from party i), such that the only possible information flow between them all is $S \to S_i$. In other words, S_i gets as input ideal leakage together with necessary information from S (e.g. trapdoors, but not leakage from other parties, since S doesn't know it) and simulates leakage based on this information. S itself doesn't see anything S_i learns from the ideal functionality or simulates. For a more formal treatment, see [BCH12].

Our simulation is corruption oblivious. Each internal state of the party (i.e. $r^*_{\mathsf{com},i}, r^*_{B,i}$) can be simulated by a subroutine S_i which gets from S a trapdoor to open HBC commitment, the program $\mathsf{ExplainB}$, and communication $a^*_1, \ldots, a^*_n$, b^*_i. S_i can first set $r^*_{\mathsf{com},i}$ by opening the commitment appropriately, and then it can generate $r^*_{B,i} \leftarrow \mathsf{ExplainB}((i, x_i, r_{\mathsf{com},i}, a^*_1, \ldots, a^*_n); b^*_i; \rho)$ for random ρ.

4.1 An Overview of the Hybrids

Here we present an overview of the hybrids. The full proof with security reductions is in section 4.2.

We start with a real execution, where $r^*_{\mathsf{com},i}, r^*_{B,i}, r^*_{\mathsf{Gen}}$ are randomly chosen, each a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$, $(\mathsf{B}, \mathsf{Eval}) \leftarrow \mathsf{Gen}(r^*_{\mathsf{Gen}})$, $b^*_i \leftarrow \mathsf{B}(i, x^*_i, r^*_{\mathsf{com},i}, a^*_1 \ldots, a^*_n; r^*_{B,i})$, $y^* \leftarrow G_k(a^*_1, \ldots, a^*_n)$.

Hybrid 1: We make challenge programs B, Eval, and $\mathsf{ExplainB}$ independent of Gen: Namely, we choose internal keys of $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$, as well as their obfuscation randomness, at random (instead of generating these values by running Gen). In addition, r^*_{Gen} is now a simulated randomness such that $\mathsf{Gen}(r^*_{\mathsf{Gen}})$ outputs $\mathsf{B}, \mathsf{Eval}$ via the trapdoor branch (instead of r^*_{Gen} being randomly chosen). Indistinguishability holds by selective indistinguishability of source and explanation for program Gen (sec. B).

Hybrid 2: We make randomness for challenge ciphertexts b^*_i independent of B: Namely, we use randomness $\mathsf{prg}(u^*_i)$, where u^*_i is chosen at random (instead of u^*_i being computed according to B). In addition, $r^*_{B,i}$ is now a simulated randomness such that $\mathsf{B}(i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n; r^*_{B,i})$ outputs b^*_i via the trapdoor branch (instead of $r^*_{B,i}$ being randomly chosen). Indistinguishability holds by selective indistinguishability of source and explanation for program B (sec. B).

This modification is done for every party.

Hybrid 3: For every party i we switch randomness used to generate challenge b^*_i from $\mathsf{prg}(u^*_{B,i})$ to truly random $\tilde{u}^*_{B,i}$, by security of a prg.

Hybrid 4: We modify programs B, Eval so that they only use a punctured version of a PRE key $K\{\{b^*_i\}^n_{i=1}\}$ and a PRF key $k\{(a^*_1, \ldots, a^*_n)\}$ (see fig. 6. Note that K is punctured at several points, while k is punctured at a single point $(a^*_1, \ldots, a^*_n)$). We don't change functionality of these programs and rely on security of iO.

In program B we can puncture the key K directly (since challenge ciphertexts use truly random $\tilde{u}^*_{B,i}$ as randomness for encryption, and since B always computes randomness as $\mathsf{prg}(u^*_i)$, the program never tries

to compute a ciphertext with challenge randomness $\tilde{u}^*_{B,i}$; by correctness of a punctured PRE key, this key correctly computes ciphertexts with randomness different from randomness used for puncturing, i.e. $\tilde{u}^*_{B,i}$).

Eval is modified as follows: if it gets as input the challenge set $(b^*_1, \ldots, b^*_n)$, then it just outputs hardwired y^*. If none of the input ciphertext is a challenge ciphertext, then it just uses a punctured key $K\{\{b^*_i\}^n_{i=1}\}$ to do its normal computation (by correctness of a PRE punctured key, these ciphertexts are decrypted correctly). The only difference is that it uses punctured PRF key $k\{(a^*_1, \ldots, a^*_n)\}$ to compute randomness R for the computation. (If it happened that b's decrypted to the challenge set $a^*_1, \ldots, a^*_n$, then the program outputs hardwired y^*, if consistency checks pass. Recall that honestly generated $\{a^*_i\}^n_{i=1}$ completely define all inputs and randomness of the computation, therefore y^* is the only non-$\perp$ output in this case). Thus the evaluation of both punctured keys on punctured inputs is avoided.

The question is what to do in Eval when some inputs are challenge ciphertexts and some are not. We claim that in this case the program should output either y^* or $\perp$ (but cannot output a different $y' \neq y^*$): indeed, since at least one of the ciphertexts is a challenge ciphertext, it contains challenge $a^*_1, \ldots, a^*_n$, and by statistical binding of an honest-but-curious commitment, each a^*_i can be verified only for x^*_i. R is completely determined by $(a_1, \ldots, a_n)$ too; thus Eval can only output $y^* = f(x^*_1, \ldots, x^*_n; R^*)$ or $\perp$. Therefore we modify the program as follows: we decrypt only non-challenge ciphertexts, and compare their $a_1, \ldots, a_n$ with challenge $a^*_1, \ldots, a^*_n$. In addition, we check that their openings of commitments are correct. If these checks pass, we output hardwired y^*, otherwise $\perp$.

Hybrid 5: We switch each ciphertext b^*_i from a real ciphertext encrypting $(i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n)$ to a simulated one. At the same time we switch the PRE key from the real punctured key to the simulated punctured key. Indistinguishability holds by the simulation security of a PRE with the punctured key.

Hybrid 6: We exploit the computational hiding property of an equivocal honest-but-curious commitment scheme and switch commitments a^*_i to simulated, together with commitment randomness $r^*_{\mathsf{com},i}$, for each party.

Hybrid 7: Finally, using security of a PRF G with punctured key $k\{(a^*_1, \ldots, a^*_n)\}$, we switch randomness R^* from $G_k(a^*_1, \ldots, a^*_n)$ to truly random value, thus making the output $y^* = f(x^*_1, \ldots, x^*_n; R^*)$ independent of our programs.

At this point the transcript can be simulated by a simulator who might not know inputs during the execution of the protocol (and only gets them upon corruption of a party), but knows the output, as explained in the beginning of the proof. Namely, commitments a^*_i and ciphertexts b^*_i are simulated; $\mathsf{Eval}, \mathsf{B}, \mathsf{ExplainB}$ are programs generated by the simulator using the PRE key $K\{\{b^*_i\}^n_{i=1}\}$, PRF key $k\{(a^*_1, \ldots, a^*_n)\}$. Hardwired variables inside programs $\mathsf{B}, \mathsf{Eval}$ are $\{a^*_i\}^n_{i=1}$, $\{b^*_i\}^n_{i=1}, y^*$, which are all known to the simulator at the end of round 1; thus, $\mathsf{Eval}, \mathsf{B}, \mathsf{ExplainB}$, and therefore r^*_{Gen} and each $r^*_{\mathsf{Gen},i}$, can be simulated. Internal state of the party can be generated by opening the commitment and by running $\mathsf{ExplainB}$ to get randomness consistent with simulated $\mathsf{Eval}, \mathsf{B}, \mathsf{ExplainB}$.

4.2 The Full Proof.

Next we present the full description of hybrids and security reductions.

Real execution

Here all variables are generated as in a real execution of the protocol, i.e.:

- **CRS generation:**
  1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
  2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}], \mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from Fig. 3.
  3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.
- **Generation of communication in the protocol:**
  1. For all i $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. For all i $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $r^*_{\mathsf{Gen}} \leftarrow \bigoplus r^*_{\mathsf{Gen},i}$ and $\mathsf{B}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K], \mathsf{Eval}[K, k], \mathsf{ExplainB}[f_{\mathsf{B}}] \leftarrow \mathsf{Gen}(r^*_{\mathsf{Gen}})$;
  4. For all i $r^*_{B,i}$ is chosen at random and $b^*_i \leftarrow \mathsf{B}(i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n; r^*_{B,i})$;
  5. For every party i $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message and b^*_i to be the second message of this party.
- **Internal state**
  1. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

In addition, y^* is computed as $f(x^*_1, \ldots, x^*_n; G_k(a^*_1, \ldots, a^*_n))$ (in case of f being deterministic, PRF-generated randomness is ignored).

Hybrid 1

In this hybrid $(\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB})$ are real-world programs generated using freshly sampled keys (as opposed to using keys produced by an extractor inside Gen). In addition, r^*_{Gen}, instead of being chosen at random, is fake randomness explaining these $(\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB})$. Each $r^*_{\mathsf{Gen},i}$ are set to sum up to r^*_{Gen}, i.e. first $n-1$ $r^*_{\mathsf{Gen},i}$ are chosen at random and the last one is set such that $r^*_{\mathsf{Gen}} = \bigoplus r^*_{\mathsf{Gen},i}$.

Indistinguishability holds because of selective indistinguishability of explanations and source (sec. B) for program Gen, i.e. $(\mathsf{Gen}, \mathsf{ExplainGen}, \mathsf{output} \leftarrow \mathsf{Gen}(\rho), \rho) \sim (\mathsf{Gen}, \mathsf{ExplainGen}, \mathsf{output} \leftarrow \mathsf{Gen}'(u), r_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}(\mathsf{output}))$ for some random u, where Gen' is a program which takes randomness u, uses it to sample keys and obfuscation randomness and outputs obfuscated programs. Note that all variables here can be generated at a CRS generation stage (in particular, nothing depends on inputs) and therefore it is enough to use selective explainability.

- **CRS generation:**
  1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
  2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}], \mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
  3. u^*_{Gen} is generated at random and used to sample keys $\mathsf{Ext}_{\mathsf{B}}$, K, k, f_{B} and obfuscation randomness for B, Eval, $\mathsf{ExplainB}$
  4. $\mathsf{B}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K], \mathsf{Eval}[K, k], \mathsf{ExplainB}[f_{\mathsf{B}}]$ are generated and obfuscated using keys from previous step (see the code in fig. 3).
  5. ρ^*_2 is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}; \rho^*_2)$;

6. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.

- **Generation of communication in the protocol:**
  1. For all i $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. For all i $r^*_{\mathsf{Gen},i}$ is chosen at random, except chronologically the last one which is set so that $r^*_{\mathsf{Gen}} = \bigoplus r^*_{\mathsf{Gen},i}$.
  3. For all i $r^*_{B,i}$ is chosen at random and $b^*_i \leftarrow \mathsf{B}(i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n; r^*_{B,i})$;
  4. For every party i $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message and b^*_i to be the second message of this party.
- **Internal state**
  1. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

Hybrid 2

In this hybrid for every party i $u^*_{B,i}$ is chosen at random, instead of applying $F_{\mathsf{Ext}_\mathsf{B}}$ to the input of program B, and b^*_i is set to be $\mathsf{PRE.Enc}_K(M^*_i; \mathsf{prg}(u^*_{B,i}))$ (where M^*_i is non-random input to program B, i.e.$(i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n)$). In addition, $r^*_{B,i}$, instead of being chosen at random, is fake randomness explaining these b^*_i on input M^*_i.

Indistinguishability is proven by invoking, one by one for each party i, selective indistinguishability of explanations and source (sec. B) for program B, i.e. $(\mathsf{B}, \mathsf{ExplainB}, b_i \leftarrow \mathsf{B}(M; r_{B,i}), r_{B,i}) \sim (\mathsf{B}, \mathsf{ExplainB}, b_i \leftarrow \mathsf{PRE.Enc}(M_i; \mathsf{prg}(u_{B,i})), r_{B,i} \leftarrow \mathsf{ExplainB}(M, b))$, for random $u_{B,i}$. Note that programs $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$ need to be determined only with the last message in round 1 (i.e. when r_{Gen} becomes defined), but by this point all inputs are already chosen by the environment. Therefore it is enough to use selective explainability.

- **CRS generation:**
  1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
  2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}], \mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
  3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.
- **Generation of chronologically first $n-1$ messages in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of the last message in round 1:**
  1. u^*_{Gen} is generated at random and used to sample keys Ext_B, K, k, f_B and obfuscation randomness for $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$;
  2. $\mathsf{B}[\mathsf{Ext}_\mathsf{B}, f_\mathsf{B}, K], \mathsf{Eval}[K, k], \mathsf{ExplainB}[f_\mathsf{B}]$ are obfuscated programs defined in fig. 3.
  3. ρ^*_2 is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}; \rho^*_2)$;

4. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
5. $r^*_{\mathsf{Gen},i}$ is set to be $\bigoplus_{i \neq j} r^*_{\mathsf{Gen},j} \oplus r^*_{\mathsf{Gen}}$.
6. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.

- **Generation of round 2 messages:**
  1. Set $M^*_i \leftarrow (i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n)$;
  2. For all i $u^*_{B,i}$ is chosen at random and $b^*_i \leftarrow \mathsf{PRE.Enc}_K(M^*_i; \mathsf{prg}(u^*_{B,i}))$;
  3. For every party i b^*_i is set to be the second message of this party.
- **Internal state**
  1. $r^*_{B,i} \leftarrow \mathsf{ExplainB}(M^*_i; b^*_i; \rho_3)$ for random ρ_3;
  2. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

Hybrid 3

In this hybrid instead of using $\mathsf{prg}(u^*_{B,i})$ as randomness when computing b^*_i, we use a random value $\tilde{u}^*_{B,i}$. Indistinguishability holds by an invocation of security of prg for every i.

- **CRS generation:**
  1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
  2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}]$, $\mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
  3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.
- **Generation of chronologically first $n-1$ messages in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of the last message in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. u^*_{Gen} is generated at random and used to sample keys $\mathsf{Ext}_{\mathsf{B}}$, K, k, f_{B} and obfuscation randomness for B, Eval, $\mathsf{ExplainB}$;
  3. $\mathsf{B}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K]$, $\mathsf{Eval}[K, k]$, $\mathsf{ExplainB}[f_{\mathsf{B}}]$ are obfuscated programs defined in fig. 3.
  4. ρ^*_2 is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}; \rho^*_2)$;
  5. $r^*_{\mathsf{Gen},i}$ is set to be $\bigoplus_{i \neq j} r^*_{\mathsf{Gen},j} \oplus r^*_{\mathsf{Gen}}$.
  6. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.

- **Generation of round 2 messages:**
  1. Set $M_i^* \leftarrow (i, x_i^*, r_{\mathsf{com},i}^*, a_1^*, \ldots, a_n^*)$;
  2. For all i $\tilde{u}_{B,i}^*$ is chosen at random and $b_i^* \leftarrow \mathsf{PRE.Enc}_K(M_i^*; \tilde{u}_{B,i}^*)$;
  3. For every party i b_i^* is set to be the second message of this party.
- **Internal state**
  1. $r_{B,i}^* \leftarrow \mathsf{ExplainB}(M_i^*; b_i^*; \rho_3)$ for random ρ_3;
  2. For every i $(r_{\mathsf{com},i}^*, r_{B,i}^*)$ is secret randomness of party i.

Hybrid 4

In this hybrid we generate different programs $\mathsf{Eval1}, \mathsf{B1}$, defined in fig. 6. The goal of this hybrid is to use a punctured PRE key $K\{\{b_i^*\}_{i=1}^n\}$ (punctured at n points) and a punctured PRF key $k\{(a_1^*, \ldots, a_n^*)\}$ (punctured at a single point) while preserving the functionality of both programs.

In program $\mathsf{B1}$ the key is punctured without any other modifications: note that $\tilde{u}_{B,i}^*$ is random, thus it doesn't have a prg preimage and no input to the program B requires computing PRE with randomness $\tilde{u}_{B,i}^*$.

In program $\mathsf{Eval1}$ significant modifications are made.

First, there is a check (case 0 on the figure), which says that if any b_j^* appears in the wrong place of the input, i.e. if any b_i in the input is equal to b_j^* for $j \neq i$, the program immediately outputs $\perp$. Note that the original program does the same, since b_j^* would decrypt to an index j, not i, and the format of i-th plaintext would be wrong. Thus after this check is made we can assume that if b_i^* is a part of the input, it is at position i.

Second, on input $b_1^*, \ldots, b_n^*$ $\mathsf{Eval1}$ outputs y^* (note that original Eval does the same).

Third, on a mixed input (when some $b_i = b_i^*$, but not all) $\mathsf{Eval1}$ does the following: it decrypts $b_i \neq b_i^*$, but doesn't decrypt b_i^*. Then it runs the code of original Eval by doing consistency checks, but only for $b_i \neq b_i^*$: it checks if the sets of $(a_1, \ldots, a_n)$ are the same in all b's (since some of b_i are b_i^*, each a_i should be equal to a_i^* inside each decrypted b), and it checks the commitment (note that there is no need in commitment check for hardcoded values since they are generated honestly and thus always pass the check). If consistency checks pass, it outputs hardcoded y^*. To see why it doesn't change the functionality of Eval, note two things. First, in this case Eval and $\mathsf{Eval1}$ output $\perp$ simultaneously. Second, if they output non-$\perp$, their output could be only y^*: this is because as long as there is at least one b_i^* in the input, all input b's has to encrypt $a_1^*, \ldots, a_n^*$, and, due to statistical binding property of the honest-but-curious commitment, verification could pass only if the set of inputs in b's is $x_1^*, \ldots, x_n^*$. In addition, $a_1^*, \ldots, a_n^*$ completely determine randomness for the computation. Thus the result will always be $f(x_1^*, \ldots, x_n^*; G_k(a_1^*, \ldots, a_n^*))$ (or $\perp$).

Finally, in the last case (when all $b_i \neq b_i^*$), the program just executes original Eval (with punctured PRE keys), with the difference that if each $a_i = a_i^*$, then the output should be y^* (again, this is because $a_1^*, \ldots, a_n^*$ completely fix inputs and randomness and therefore determine the computation). After this "if", k can be safely punctured at $(a_1^*, \ldots, a_n^*)$.

It is easy to see that PRE decryption is never invoked on any b_i^*, and PRF G is never called on $(a_1^*, \ldots, a_n^*)$, and therefore both keys can be punctured.

Indistinguishability holds by iO.

- **CRS generation:**
  1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
  2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}]$, $\mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
  3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.
- **Generation of chronologically first $n-1$ messages in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $\tilde{u}^*_{B,i}$ is chosen at random;
  4. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of the last message in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. $\tilde{u}^*_{B,i}$ is chosen at random;
  3. Set $M^*_i \leftarrow (i, x^*_i, r^*_{\mathsf{com},i}, a^*_1, \ldots, a^*_n)$;
  4. For all i $b^*_i \leftarrow \mathsf{PRE.Enc}_K(M^*_i; \tilde{u}^*_{B,i})$;
  5. u^*_{Gen} is generated at random and used to sample keys $\mathsf{Ext}_{\mathsf{B}}$, K, k, f_{B} and obfuscation randomness for B, Eval, $\mathsf{ExplainB}$;
  6. R^* is set to be $G_k(a^*_1, \ldots, a^*_n)$;
  7. y^* is set to be $f(x^*_1, \ldots, x^*_n; R^*)$;
  8. $\mathsf{B1}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K\{\{b^*_i\}^n_{i=1}\}]$, $\mathsf{Eval1}[K\{\{b^*_i\}^n_{i=1}\}, k\{(a^*_1, \ldots, a^*_n)\}, y^*]$, $\mathsf{ExplainB}[f_{\mathsf{B}}]$ are obfuscated programs defined in fig. 6.
  9. ρ^*_2 is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B1}, \mathsf{Eval1}, \mathsf{ExplainB}; \rho^*_2)$;
  10. $r^*_{\mathsf{Gen},i}$ is set to be $\bigoplus_{i \neq j} r^*_{\mathsf{Gen},j} \oplus r^*_{\mathsf{Gen}}$.
  11. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of round 2 messages:**
  1. For every party i b^*_i is set to be the second message of this party.
- **Internal state**
  1. $r^*_{B,i} \leftarrow \mathsf{ExplainB}(M^*_i; b^*_i; \rho_3)$ for random ρ_3;
  2. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

Programs used in the proof and the simulation

Program $\mathsf{B1}(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n; r_{B,i})$
Constants: an extracting PRF key Ext_B, faking PDE key f_B, punctured PRE key $K\{\{b_i^*\}_{i=1}^n\}$

- **Trapdoor branch:**
  1. set $(i', x', r'_{\mathsf{com},i}, a'_1, \ldots, a'_n, b', \tilde{\rho}) \leftarrow \mathsf{PDE.Dec}_{f_\mathsf{B}}(r_{B,i})$. If decryption returns $\perp$ then goto normal branch;
  2. if $(i', x', r'_{\mathsf{com},i}, a'_1, \ldots, a'_n) \neq (i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$ then goto normal branch;
  3. output b' and halt;
- **Normal branch:**
  1. Set $M = (i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
  2. $u_{B,i} \leftarrow F_{\mathsf{Ext}_\mathsf{B}}(M, r_{B,i})$
  3. Set $b \leftarrow \mathsf{PRE.Enc}_{K\{\{b_i^*\}_{i=1}^n\}}(M; \mathsf{prg}(u_{B,i}))$
  4. Output b

Program $\mathsf{Eval1}(b_1, \ldots, b_n)$
Constants: punctured PRE key $K\{\{b_i^*\}_{i=1}^n\}$, punctured PRF key $k\{(a_1^*, \ldots, a_n^*)\}$, $a_1^*, \ldots, a_n^*$, $b_1^*, \ldots, b_n^*, y^*$
Case 0: If there is $i \neq j$ such that $b_i = b_j^*$, output $\perp$.
Case 1: If for all i $b_i = b_i^*$, then output y^* and halt.
Case 2: If for some i $b_i = b_i^*$ (denote such set as $\mathcal{I}$), then:

1. For every $i \notin \mathcal{I}$ decrypt:
   (a) Set $M_i \leftarrow \mathsf{PRE.Dec}_{K\{\{b_i^*\}_{i=1}^n\}}(b_i)$;
   (b) Parse M_i as $(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
2. For every $i \notin I$ check consistency:
   (a) Verify that the set $(a_1, \ldots, a_n)$ is the same as $(a_1^*, \ldots, a_n^*)$
   (b) Verify that $a_i = \mathsf{HBCCommit}(i, x_i; r_{\mathsf{com},i})$
3. Output y^*.

Case 3: If for all i $b_i \neq b_i^*$, then:

1. For every i decrypt:
   (a) Set $M_i \leftarrow \mathsf{PRE.Dec}_{K\{\{b_i^*\}_{i=1}^n\}}(b_i)$;
   (b) Parse M_i as $(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n)$
2. For every i check consistency:
   (a) Verify that the set $(a_1, \ldots, a_n)$ is the same in all $M_1, \ldots, M_n$;
   (b) Verify that $a_i = \mathsf{HBCCommit}(i, x_i; r_{\mathsf{com},i})$
3. If $(a_1, \ldots, a_n) = (a_1^*, \ldots, a_n^*)$ then output y^*
4. Set $R \leftarrow G_{k\{(a_1^*, \ldots, a_n^*)\}}(a_1, \ldots, a_n)$.
5. Output $y \leftarrow f(x_1, \ldots, x_n; R)$.

Figure 6: Programs used in the proof and the simulation.

Hybrid 5

In this hybrid we switch each b_i^* from a real PRE ciphertext to a simulated ciphertext. In addition, we switch the punctured PRE key $K\{\{b_i^*\}_{i=1}^n\}$ to simulated.

Indistinguishability holds by security of PRE (def. 1).

- **CRS generation:**
  1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
  2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}]$, $\mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
  3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.
- **Generation of chronologically first $n-1$ messages in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of the last message in round 1:**
  1. $r^*_{\mathsf{com},i}$ is chosen at random and a^*_i is set to $\mathsf{HBCCommit}(i, x^*_i; r^*_{\mathsf{com},i})$;
  2. u^*_{Gen} is generated at random and used to sample keys $\mathsf{Ext}_{\mathsf{B}}$, k, f_{B} and obfuscation randomness for $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$;
  3. R^* is set to be $G_k(a^*_1, \ldots, a^*_n)$;
  4. y^* is set to be $f(x^*_1, \ldots, x^*_n; R^*)$;
  5. $K\{\{b^*_i\}^n_{i=1}\}, b^*_1, \ldots, b^*_n$ are generated by a simulator for the PRE.
  6. $\mathsf{B1}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K\{\{b^*_i\}^n_{i=1}\}]$, $\mathsf{Eval1}[K\{\{b^*_i\}^n_{i=1}\}, k\{(a^*_1, \ldots, a^*_n)\}y^*]$, $\mathsf{ExplainB}[f_{\mathsf{B}}]$ are obfuscated programs defined in fig. 6.
  7. ρ^*_2 is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B1}, \mathsf{Eval1}, \mathsf{ExplainB}; \rho^*_2)$;
  8. $r^*_{\mathsf{Gen},i}$ is set to be $\bigoplus_{i \neq j} r^*_{\mathsf{Gen},j} \oplus r^*_{\mathsf{Gen}}$.
  9. $(a^*_i, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of round 2 messages:**
  1. For every party i b^*_i is set to be the second message of this party.
- **Internal state**
  1. $r^*_{B,i} \leftarrow \mathsf{ExplainB}(M^*_i; b^*_i; \rho_3)$ for random ρ_3;
  2. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

Hybrid 6

In this hybrid we switch each a^*_i from a real honest-but-curious commitment to a simulated commitment. In addition, we switch its randomness $r^*_{\mathsf{com},i}$ to simulated.

Security holds by computational hiding of the equivocal honest-but-curious commitment (def. 2).

- **CRS generation:**

1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}]$, $\mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.

- **Generation of chronologically first $n-1$ messages in round 1:**
  1. a_i^* is a simulated commitment of an honest-but-curious commitment
  2. $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $(a_i^*, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of the last message in round 1:**
  1. a_i^* is a simulated commitment of an honest-but-curious commitment
  2. u^*_{Gen} is generated at random and used to sample keys $\mathsf{Ext}_{\mathsf{B}}$, k, f_{B} and obfuscation randomness for $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$;
  3. R^* is set to be $G_k(a_1^*, \ldots, a_n^*)$;
  4. y^* is set to be $f(x_1^*, \ldots, x_n^*; R^*)$;
  5. $K\{\{b_i^*\}_{i=1}^n\}, b_1^*, \ldots, b_n^*$ are generated by a simulator for the PRE.
  6. $\mathsf{B1}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K\{\{b_i^*\}_{i=1}^n\}]$, $\mathsf{Eval1}[K\{\{b_i^*\}_{i=1}^n\}, k\{(a_1^*, \ldots, a_n^*)\}]$, $\mathsf{ExplainB}[f_{\mathsf{B}}]$ are obfuscated programs defined in fig. 6.
  7. ρ_2^* is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B1}, \mathsf{Eval1}, \mathsf{ExplainB}; \rho_2^*)$;
  8. $r^*_{\mathsf{Gen},i}$ is set to be $\bigoplus_{i \neq j} r^*_{\mathsf{Gen},j} \oplus r^*_{\mathsf{Gen}}$.
  9. $(a_i^*, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of round 2 messages:**
  1. For every party i b_i^* is set to be the second message of this party.
- **Internal state**
  1. $r^*_{B,i} \leftarrow \mathsf{ExplainB}(M_i^*; b_i^*; \rho_3)$ for random ρ_3;
  2. $r^*_{\mathsf{com},i}$ is a simulated opening of a_i^* to (i, x_i^*);
  3. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

Hybrid 7

In this hybrid we switch R^* from $G_k(a_1^*, \ldots, a_n^*)$ to a truly random value. Therefore the result of the computation is switched from $y^* = f(x_1^*, \ldots, x_n^*; G_k(a_1^*, \ldots, a_n^*))$ to $y^* = f(x_1^*, \ldots, x_n^*; R^*)$ for random R^*.

Security holds by security of a punctured PRF G.

- **CRS generation:**

1. Keys f_{Gen}, $\mathsf{Ext}_{\mathsf{Gen}}$ are sampled;
2. Programs $\mathsf{Gen}[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}]$, $\mathsf{ExplainGen}[f_{\mathsf{Gen}}]$ are obfuscated programs from fig. 3.
3. $(\mathsf{Gen}, \mathsf{ExplainGen})$ is set to be the CRS.

- **Generation of chronologically first $n-1$ messages in round 1:**
  1. a_i^* is a simulated commitment of an honest-but-curious commitment
  2. $r^*_{\mathsf{Gen},i}$ is chosen at random;
  3. $(a_i^*, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of the last message in round 1:**
  1. a_i^* is a simulated commitment of an honest-but-curious commitment
  2. R^* is set to be random;
  3. y^* is set to be $f(x_1^*, \ldots, x_n^*; R^*)$;
  4. u^*_{Gen} is generated at random and used to sample keys $\mathsf{Ext}_{\mathsf{B}}$, k, f_{B} and obfuscation randomness for $\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}$;
  5. $K\{\{b_i^*\}_{i=1}^n\}, b_1^*, \ldots, b_n^*$ are generated by a simulator for the PRE.
  6. $\mathsf{B1}[\mathsf{Ext}_{\mathsf{B}}, f_{\mathsf{B}}, K\{\{b_i^*\}_{i=1}^n\}]$, $\mathsf{Eval1}[K\{\{b_i^*\}_{i=1}^n\}, k\{(a_1^*, \ldots, a_n^*)\}, y^*]$, $\mathsf{ExplainB}[f_{\mathsf{B}}]$ are obfuscated programs defined in fig. 6.
  7. ρ_2^* is chosen at random and $r^*_{\mathsf{Gen}} \leftarrow \mathsf{ExplainGen}[f_{\mathsf{Gen}}](\mathsf{B1}, \mathsf{Eval1}, \mathsf{ExplainB}; \rho_2^*)$;
  8. $r^*_{\mathsf{Gen},i}$ is set to be $\bigoplus_{i \neq j} r^*_{\mathsf{Gen},j} \oplus r^*_{\mathsf{Gen}}$.
  9. $(a_i^*, r^*_{\mathsf{Gen},i})$ is set to be the first message of a party i.
- **Generation of round 2 messages:**
  1. For every party i b_i^* is set to be the second message of this party.
- **Internal state**
  1. $r^*_{B,i} \leftarrow \mathsf{ExplainB}(M_i^*; b_i^*; \rho_3)$ for random ρ_3;
  2. $r^*_{\mathsf{com},i}$ is a simulated opening of a_i^* to (i, x_i^*);
  3. For every i $(r^*_{\mathsf{com},i}, r^*_{B,i})$ is secret randomness of party i.

This hybrid corresponds to the simulation described in the beginning of the proof.

5 RAM-efficient MPC Protocol Against Malicious Adversaries

In order to obtain 2-round RAM-efficient protocol secure against malicious corruption of possibly all parties, we plug a RAM-efficient NIZK into the protocol of Garg and Polychroniadou [GP14].

5.1 RAM-efficient Statistically-Sound Non-interactive Zero Knowledge

We describe a statistically-sound NIZK scheme where the work of each party, as well as the communication complexity, only depend on corresponding parameters of a relation represented as a RAM program, and not as a circuit.

Our scheme is described on figure 7. It uses a non-succinct statistically sound NIZK and a succinct garbling scheme for RAM with an additional property, which we call perfect correctness with abort: it says that for *any* choice of keys and randomness for garbling by a possibly malicious garbler, the evaluation by an honest evaluator results in either the correct value $f(x)$ or in $\perp$[16].

To prove that there exist w for which $R(x, w) = 1$, the prover sends to the verifier a garbled relation R and a garbled input (x, w), together with the proof that both garbled values were generated correctly. The verifier checks the proof and accepts if evaluation of the garbled program results in 1.

Definition 3. *A garbling scheme (*$\mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp}$*) has perfect correctness with abort, if for any randomness* r_1, r_2, r_3 *the following holds: let* $K \leftarrow \mathsf{KeyGen}(1^\lambda; r_1)$*, let* $\tilde{f} \leftarrow \mathsf{GProg}(K; f; r_2)$ *and* $\tilde{x} \leftarrow \mathsf{GInp}(K; x; r_3)$*. Then either* $\tilde{f}(\tilde{x}) = f(x)$ *or* $\tilde{f}(\tilde{x}) = \perp$*.*

Lemma 2. *The succinct garbling scheme for RAM by Canetti and Holmgren ([CH16]) satisfies correctness with abort, as defined in def. 3.*

In appendix A we outline the construction of [CH16] and explain why perfect correctness with abort holds.

Theorem 7. *Let* (GenCRS, P, V) *be a statistically sound non-interactive zero-knowledge proof system. Let* $(\mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp})$ *be a succinct garbling scheme which is perfectly correct with abort (def. 3), i.e.:*

- *The circuit complexity of* $\mathsf{KeyGen}, \mathsf{GProg}$*, and* GInp*, and therefore the sizes of a garbled input and a garbled program, only depend on* λ*,* $|x|$*,* $|f|_{\mathsf{RAM}}$*,* $\log \mathsf{Time}(f, x)$*,* $\log \mathsf{Space}(f, x)$*,*
- *The time to evaluate a garbled program* $\tilde{f}$ *on a garbled input* $\tilde{x}$ *only depends on* λ*,* $\mathsf{Time}(f, x)$*,* $\log \mathsf{Space}(f, x)$*.*

Then the scheme on figure 7 is a succinct statistically sound non-interactive zero-knowledge proof system, i.e. let T, S *be time and space upperbounds of* $R(x, w)$ *for all possible* w*. Then:*

1. *The running time of the prover (and therefore the size of the proof) only depends on* λ*,* $|x|$*,* $|w|$*,* $|R|_{\mathsf{RAM}}$*,* $\log S$*,* $\log T$*;*
2. *The running time of the verifier only depends on* λ*,* $|x|$*,* $|w|$*,* $|f|_{\mathsf{RAM}}$*,* T*,* $\log S$*.*

[16]Note that the evaluation is not allowed to always output $\perp$, since there is still a normal correctness of the garbling scheme, which says that with overwhelming probability over the choice of the key the result of the computation is correct. This of course doesn't rule out the possibility that for negligible fraction of bad keys the evaluation always outputs $\perp$.

Succinct statistically sound NIZK:

CRS generation generates the CRS of underlying NIZK as $\mu \leftarrow \mathsf{GenCRS}(1^\lambda)$ and sets μ to be a CRS.

The prover on input (μ, x, w) does the following:

1. It chooses randomness r and runs $K \leftarrow \mathsf{KeyGen}(1^\lambda; r)$ of a garbling scheme.
2. It garbles $\widetilde{R} \leftarrow \mathsf{GProg}(K; R(\cdot,\cdot); \rho_1)$ and $\widetilde{xw} \leftarrow \mathsf{GInp}(K; (x, w); \rho_2)$ using randomness ρ_1, ρ_2.
3. It generates a proof $\pi \leftarrow P(\mu; (x, \widetilde{R}, \widetilde{xw}, \mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp}); (r, \rho_1, \rho_2, w))$ for the statement "There exist $(r, \rho_1, \rho_2, K, w)$ such that $K = \mathsf{KeyGen}(1^\lambda; r)$ and $\widetilde{R} = \mathsf{GProg}(K; R(\cdot,\cdot); \rho_1)$ and $\widetilde{xw} = \mathsf{GInp}(K; (x, w); \rho_2)$".
4. It outputs $(\widetilde{R}, \widetilde{xw}, \pi)$ as a proof.

The verifier on input $(\mu, x, (\widetilde{R}, \widetilde{xw}, \pi))$ does the following:

1. It checks that $\widetilde{R}(\widetilde{xw}) = 1$;
2. It checks that the proof π is correct by running $V(\mu; (x, \widetilde{R}, \widetilde{xw}, \mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp}); \pi)$.
3. If both checks pass, it accepts. Otherwise, it rejects.

Figure 7: The description of succinct statistically sound NIZK. The scheme uses a succinct garbling scheme $(\mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp})$ with perfect correctness with abort, as well as a (non-succinct) statistically sound NIZK (GenCRS, P, V). To prove that there exist w such that $R(x, w)$ holds, the prover garbles a program $R(\cdot,\cdot)$, garbles an input (x, w), and generates a proof that garbling was done correctly (i.e. according to programs $\mathsf{GProg}, \mathsf{GInp}$, and that correct x and R were garbled). The verifier accepts if and only proofs verify and the evaluation of a garbled program on a garbled input gives 1.

Now we give a proof of the theorem:

Proof. Completeness of the scheme directly follows from completeness of underlying NIZK and correctness of a garbling scheme.

Statistical soundness. By statistical soundness of the underlying NIZK, and by perfect correctness with abort of the garbling scheme, if the verifier accepts the proof π of underlying NIZK (check 2 on fig. 7), then $\widetilde{R}(\widetilde{xw}) = R(x, w)$ or $\perp$. Thus, if the verifier accepts the proof $(\widetilde{R}, \widetilde{xw}, \pi)$ of the succinct NIZK, then $\widetilde{R}(\widetilde{xw}) = R(x, w) = 1$, i.e. $x \in L$.

Computational zero-knowledge. We simulate the proof using the simulators Sim_G of the garbling scheme and $\mathsf{Sim}_{\mathsf{NIZK}}$ of underlying NIZK.

First the simulator runs $\mathsf{Sim}_{\mathsf{NIZK}}$ to generate a simulated CRS μ. On input x the simulator simulates the proof by running $\mathsf{Sim}_G(1, |x|, |w|, |R|, T, S)$ to generate simulated garbled values $\widetilde{R}, \widetilde{xw}$. Then it runs $\mathsf{Sim}_{\mathsf{NIZK}}(x, \widetilde{R}, \widetilde{xw}, \mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp})$ to generate a simulated proof for the false statement that $\widetilde{R}, \widetilde{xw}$ were generated properly. Indistinguishability of a real and simulated proof holds by invocation of security of a garbling scheme and zero-knowledge of underlying NIZK.

Succinctness. It can be immediately verified that the running time of the prover and the verifier satisfies the statement of the theorem. □

Acknowledgments

We thank Justin Holmgren for pointing out that our MPC protocol can be used to compute a garbling scheme in [IK02] manner, which allows us to avoid the use of subexponentially-secure iO even in the RAM setting.

References

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Computationally private randomizing polynomials and their applications. *Computational Complexity*, 15(2):115–162, 2006. 4

[BCH12] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, pages 266–284, 2012. 1, 2, 21

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 742–762, 2015. 4

[BST14] Mihir Bellare, Igors Stepanovs, and Stefano Tessaro. Poly-many hardcore bits for any one-way function and a framework for differing-inputs obfuscation. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 102–121, 2014. 12, 13, 17

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 61–85, 2007. 1

[CGP15] Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. Adaptively secure two-party computation from indistinguishability obfuscation. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 557–585, 2015. 1, 4, 6

[CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 169–178, 2016. 4, 5, 10, 11, 18, 32, 35, 36

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 429–437, 2015. 4

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503, 2002. 2

[CP13] Kai-Min Chung and Rafael Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013. 36

[DKR14] Dana Dachman-Soled, Jonathan Katz, and Vanishree Rao. Adaptively secure, universally composable, multi-party computation in constant rounds. *IACR Cryptology ePrint Archive*, 2014:858, 2014. 1, 4, 5, 6, 17, 37

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, pages 144–163, 2011. 4

[Gen09] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729. 5

[GGHR14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. Two-round secure MPC from indistinguishability obfuscation. In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pages 74–94, 2014. 4

[GP14] Sanjam Garg and Antigoni Polychroniadou. Two-round adaptively secure MPC from indistinguishability obfuscation. *IACR Cryptology ePrint Archive*, 2014:844, 2014. 1, 3, 4, 5, 6, 10, 31

[Gro11] Jens Groth. Minimizing non-interactive zero-knowledge proofs using fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2011:12, 2011. 5

[IK02] Yuval Ishai and Eyal Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, pages 244–256, 2002. 4, 34

[NY90] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 427–437, 1990. 6

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 475–484, 2014. 5, 11, 12, 13, 17, 37

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 678–697, 2015. 8, 17

A An Overview of The Garbling Scheme by Canetti and Holmgren

In this section we explain why the garbling scheme from [CH16] has perfect correctness with abort, i.e. that for any choice of the key and randomness an honest evaluation results in either correct $f(x)$ or $\perp$. This property is needed for our construction of a succinct NIZK, as explained in the introduction and in section 5.1.

Definition 4. *A garbling scheme (*$\mathsf{KeyGen}, \mathsf{GProg}, \mathsf{GInp}$*) has a perfect correctness with abort, if for any randomness* r_1, r_2, r_3 *the following holds: let* $K \leftarrow \mathsf{KeyGen}(1^\lambda; r_1)$*, let* $\tilde{f} \leftarrow \mathsf{GProg}(K; f; r_2)$ *and* $\tilde{x} \leftarrow \mathsf{GInp}(K; x; r_3)$*. Then either* $\tilde{f}(\tilde{x}) = f(x)$ *or* $\tilde{f}(\tilde{x}) = \perp$*.*

This definition can be thought of correctness with semi-malicious garbler, i.e. the garbler who follows the garbling instructions but chooses all randomness maliciously.

The construction of [CH16] proceeds in four steps. We briefly outline each step (omitting many details unnecessary for correctness with abort) and explain why perfect correctness with abort holds.

Step 1. Their first step is to build a garbling scheme which guarantees indistinguishability of $\widetilde{M_0}, \widetilde{x}$ and $\widetilde{M_1}, \widetilde{x}$ but only as long as the trace of the computation of $M_0(x)$ and $M_1(x)$ (i.e. the sequence of CPU states and memory operations) is exactly the same.

This is done by obfuscating a CPU step function C, which takes as input a CPU state q, a memory symbol v, and outputs $(q', \mathsf{op}') \leftarrow C(q, v)$, which are the next state and the next memory operation. This construction is augmented with a number of mechanisms (such as signatures) to make sure that a malicious evaluator cannot feed a wrong state or wrong memory symbol to the program.

We claim that this scheme has perfect correctness with abort. First, note that an honest verifier initiates the computation with the correct memory symbol (we can assume that the initial memory symbol the program expects to see is always $\perp$) and state q_{start}. Next, if the computation at steps $1, \ldots, i-1$ was correct (i.e. resulting in the same state and memory operation which one would get by executing non-garbled $f(x)$), then the output at step i also results in the correct new state and memory operation, since the honest verifier feeds in correct values at step i. Therefore by induction the result is the correct $f(x)$. Note that all additional checks in the program (possibly chosen with bad parameters) can only reject the input (and therefore halt the execution), but cannot alter q, v, and op, and therefore cannot alter an honest execution and its result.

Step 2. In this step the state becomes hidden. Instead of getting a state q in the clear and outputting it in the clear, the program outputs it encrypted. Since this encryption they use is perfectly correct, this encryption-decryption procedure cannot alter the state to make it incorrect, and therefore the perfect correctness with abort still holds.

Step 3. In this step the memory (but not the access pattern) becomes hidden: the program works with encrypted memory (instead of memory in the clear), and in particular outputs encrypted memory operations and gets as input encrypted symbols. Since this encryption is perfectly correct, this encryption-decryption procedure cannot alter the memory content, input symbols and output operations and make them incorrect; therefore the perfect correctness with abort still holds.

Step 4. The last step is to hide the access pattern. This is done by implementing a Chung-Pass ORAM ([CP13]). This ORAM itself is perfectly correct with abort, i.e. at each step either the memory operation is implemented correctly, or the overflow ($\perp$) occurs. Thus the whole construction of the garbling scheme is perfectly correct with abort.

B Explainability Compiler

The original construction of a deniable encryption by Sahai and Waters [SW14] gives a way to make a single algorithm "adaptively secure": i.e. it transforms a randomized program $\mathsf{Alg}(x; r)$ into a different one $\widetilde{\mathsf{Alg}}(x; r)$ (by adding a trapdoor branch and rerandomizing the program) so that is possible to generate fake randomness consistent with a given input and output.

The important property which we use in our proofs is *indistinguishability of source and explanation*. Roughly speaking, indistinguishability of source says that for random r $\mathsf{Alg}(x; r)$ and $\widetilde{\mathsf{Alg}}(x; r)$ are indistinguishable. Indistinguishability of explanations says that real randomness r is indistinguishable from fake randomness r which results in the same output $a = \widetilde{\mathsf{Alg}}(x; r)$. These properties combined together state that random r and the output $a = \widetilde{\mathsf{Alg}}(x; r)$ are indistinguishable from the output of original program $a = \mathsf{Alg}(x; u)$ on some random u, together with fake randomness r which makes compiled $\widetilde{\mathsf{Alg}}(x; r)$ output a. This holds even when the program to generate fake randomness is publicly available.

The way to think about indistinguishability of source and explanation is the following: it is possible to move from "a real world" (random r, $a \leftarrow \widetilde{\mathsf{Alg}}(x; r)$) to a "hybrid" where $a \leftarrow \mathsf{Alg}(x; u)$, and r is fake, but pretending to be real randomness. Essentially this step allows to "detach" a from a complicated $\widetilde{\mathsf{Alg}}$ and make it the result of a simpler Alg. Because of this detaching, in the next hybrid we could use security of the primitive realized by Alg while still being able to generate internal state r: say, if Alg is an encryption scheme, then in the next hybrid we could switch it to encryption of a different value.

We also note that this indistinguishability is only selective, i.e. the input x has to be known before the indistinguishability game can be played. This imposes some restrictions on the constructions and proofs (in particular, this is one of the reasons why we need nested programs).

Since this technique became standard in the world of adaptive security, we only briefly outlined it here. For formal definitions, constructions, and proofs, we refer the reader to the paper of Dachman-Soled et al ([DKR14]) who formalized the technique under the name of explainability compiler.