Better Two-Round Adaptive Multi-party Computation

Ran Canetti[1,2], Oxana Poburinnaya[1](✉),
and Muthuramakrishnan Venkitasubramaniam[3]

[1] Boston University, Boston, USA
{canetti,oxanapob}@bu.edu
[2] Tel Aviv University and CPIIS, Tel Aviv, Israel
[3] University of Rochester, Rochester, USA
muthuv@cs.rochester.edu

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 than GP, we show a two-round, adaptively secure protocol where:

- Only a global (i.e., non-programmable) reference string is needed. In contrast, in GP the reference string is programmable, even in the semi-honest case.
- Only *polynomially-secure* indistinguishability obfuscation for circuits and injective one way functions are assumed. In GP, sub-exponentially secure IO is assumed.

Second, we show how to make the GP protocol have only RAM complexity, even for Byzantine corruptions. For this we construct the first *statistically-sound* non-interactive Zero-Knowledge scheme with RAM complexity.

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 is of central interest. For one, in the case of protocols for computing randomized functions, it allows requiring that

Research supported by the NSF MACS Frontier project, ISF grant 1523/14, Google Faculty Research Grant and NSF Awards CNS-1526377/1618884.

S. Fehr (Ed.): PKC 2017, Part II, LNCS 10175, pp. 396–427, 2017.
DOI: 10.1007/978-3-662-54388-7_14

the internal randomness of the function remains hidden even when the entire state of the system is exposed. 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 *two* rounds (which is the minimum possible).

We construct better two-round, fully adaptive protocols for general multiparty computation. Our improvements span a number of security, functionality, and efficiency aspects. We start by presenting and discussing some of these aspects.

Randomness-Hiding Functionalities. Consider a set S of parties that want to run a secure function evaluation protocol in order to jointly generate an obfuscated program, where the program is to be used in some other protocol that involves additional parties. Security of the obfuscated program should be preserved even when everybody in the set S is corrupted (which could be important for the remaining honest parties in the other protocol). Note that this program-obfuscating functionality is randomized, and security of the overall system requires that the randomness of this function remains secret even when all parties in S are corrupted. Another example of such a task is to instruct parties to joinly sample an RSA public key $N = pq$ without knowing the actual factorization p, q, even when the secret information of all parties is pooled together. We call protocols that hide the actual randomness which was used to compute the function even when everybody is corrupted *randomness-hiding.*

We note that the standard methodology of evaluating a randomized functionality via secure evaluation of a circuit, where some of the input values to the circuit are the result of xor-ing the local random inputs of all parties, results in a protocol that is inherently not randomness-hiding[1]. With this approach the adversary corrupting everybody learns the randomness of each and every party, and therefore the internal randomness of the function (e.g. random coins of obfuscation); thus no security is left.

Randomness hiding is also useful in another, perhaps less obvious, scenario. Adaptive security is often used to argue leakage tolerance [BCH12]: assume parties are computing a randomized functionality, and the adversary decides to leak 1 bit of each party's randomness. If the protocol looses security when everybody is corrupted, the simulator from [BCH12] cannot simulate such leakage,

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

since the argument from [BCH12] requires that the simulator should be able to potentially simulate *the full randomness of each party whose internal state was leaked,* even though the adversary actually sees only a single bit of randomness of each party.[2] In contrast, if the protocol supports randomness-hiding functionalities, then the simulator can simulate randomness of all parties, and therefore the protocol remains leakage-tolerant even if the adversary decides to leak from everybody.

Global Common Reference String. In the common reference string (CRS) model, all parties have access to a string, generated in advance by a trusted entity (which doesn't need to participate in the protocol). In a *local* (sometimes called *programmable*) CRS model, which is most often used, the simulator has the power to generate the CRS itself. This makes the task of designing protocols easier, since the simulator can generate the CRS in such a way that it knows corresponding trapdoors and therefore has more power than the adversary. The major drawback of a local CRS is that when two different protocols use the same CRS, *there is no guarantee of security whatsoever*, even if each of them separately is secure. Thus, 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.

To overcome these issues with composability, the *global* CRS model was introduced. In this model the simulator doesn't have the power to generate the CRS; instead, it has to work with a given CRS. 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 this sence composition-wise *the global CRS model is very close to the plain model*: once we proved that the protocol is secure with a global CRS, we don't need to take this CRS into account anymore, since it can be used by any other protocol without the risk of compromising security.

On the Need of the Common Reference String. Our protocol works in a common reference string (CRS) model. While there is no evidence that computing randomness-hiding functionalities require a CRS[3], it is not known how to compute general randomness-hiding functionalities in the plain model. In fact, this is an interesting open problem, and solving it would allow to remove the CRS requirement from many works (including this work), where the CRS is an obfuscated program whose keys and randomness should remain hidden.

[2] To be more precise, [BCH12] require that there exist a translation function which maps ideal world internal state into real world internal state.

[3] Indeed, some simple functions can be computed in a randomness-hiding way even in the plain model; for instance, the function $f(r) = g^r$, where g is a group generator and r is randomness, can be simply computed by choosing a random element in a group; in this case randomness r remains unknown.

As discussed in [IKOS10], adaptively secure protocols for randomized functionalities are tightly connected to extractable one way functions (EOWF). Namely, this work shows that the existence of such a protocol for general functionalities *in the plain model* implies that EOWFs with uniform auxiliary input don't exist, since one-wayness of the function can be broken by first using the simulator to obtain random coins for a given output and then by running the extractor on these random coins to extract the actual input of the EOWF.

We also stress that the CRS appears to be essential, even in the semi-honest setting. Recall that in the case of non-adaptive semi-honest security, CRS is not needed; indeed, instead of having a CRS, parties can generate the CRS by themselves, in the plain model, in the beginning of the protocol, at the cost of one more round. However, this is not true in the case of adaptive security. The reason is that our CRS contains secrets (e.g. randomness of the obfuscation, PRF keys) which shouldn't be known to anybody, including parties running the protocol. Working in the plain model would require parties to generate this CRS in a way that even all parties together do not know corresponding secrets. As discussed in the previous paragraph, this is an open problem.

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.)

Although we cannot take advantage of a potentially sublinear RAM computations (indeed, unlike, say, the persistent garbled RAM setting where database garbling phase could be long, but the actual computations are very short, the MPC setting requires the computation to touch every input), multiparty computation can still benefit from the RAM model in several ways. As one example, consider the case where parties are willing to trade some security for efficiency; in this case 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

[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.

(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 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.

On the Limitations of the [IK02, AIK06] *Approach in the Fully Adaptive Setting.* A natural approach to obtaining protocols with RAM efficiency is to use ideas of [IK02, AIK06]: Instead of directly evaluating the desired function, have the parties jointly evaluate a garbling (or, randomized encoding) of the function and input. Then each party locally computes the output. Plugging-in a RAM-efficient garbling scheme [CHJV15, CH16] results in RAM-efficient protocols. However, this approach has a caveat in our fully adaptive setting: note that the functionality which needs to be computed (i.e. garbling) is randomized. If we want to achieve full adaptive security, the randomness used in the garbling should remain hidden even when everybody is corrupted; in other words, for the whole construction to be secure, the underlying protocol should be randomness-hiding. However, the only two-round protocol with full adaptive security we know (that of [GP14]) is not randomness-hiding, and therefore to use this approach we need to come up with adaptively secure randomness-hiding protocol first.

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. Besides globality, our protocol has other features: First, the protocol allows to securely compute even randomness-hiding functionalities, and furthermore, it guarantees leakage tolerance even when every party can be leaked from (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). Second, 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. Third, 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 semi-honest MPC with global setup[5];
– The first 2-round fully adaptive semi-honest MPC which doesn't require subexponential security of iO;
– The first 2-round fully adaptive semi-honest MPC which supports all (even randomness-hiding) functionalities, and which therefore is fully leakage tolerant.

Making this Protocol Secure Against Malicious Adversaries. The common techniques [CLOS02] can be applied to compile this protocol into its malicious version. The resulting protocol needs 4 rounds - two rounds should be added in the beginning to do a malicious coin toss by first committing to inputs and randomness and then partially opening randomness. We observe however that the first round of the semi-honest protocol is a commitment round as well, and thus in the malicious version we can use CLOS commitments as if they were round-1 messages of the semi-honest protocol. Thus, then protocol requires only three rounds (round 1 for commitments, round 2 for partial opening randomness, and round 3 for round 2 of the semi-honest protocol). The resulting protocol preserves all properties of the semi-honest version (in particular, it remains randomness-hiding as long as there is at least one uncorrupted party during round 2, which could be corrupted later). The only property that is lost is globality of the CRS, which is inherent in the malicious setting). The resulting protocol outperforms the protocol by Dachman-Soled et al. [DKR14], which is a 4-round protocol against semi-honest adversaries.

1.2 Our Results: Malicious Setting

As an additional result, we show how to make the protocol of [GP14] RAM-efficient: namely, we construct the first RAM-efficient statistically-sound non-interactive zero-knowledge proofs, and then plug this NIZK into the protocol of [GP14]. Compared to the malicious version of our first protocol, this protocol needs only two rounds (instead of three), however, it requires subexponentially-secure iO, and is not randomness-hiding.

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.*

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]):

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

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.

1.3 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 require the function to be represented as a circuit: 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,

[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.

plugging a RAM-efficient garbling scheme [CHJV15, CH16] into known constructions results in statically-secure RAM-efficient protocols. However, in order to achieve adaptive security, the underlying protocol must support randomness-hiding functionalities. Prior to our work, no fully adaptive, two round protocol with randomness hiding was known.

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. All parameters are for the semi-honest setting.

	Rounds	Supports RAM	Global CRS	Randomness hiding	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

[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.

Succinct NIZK Proofs. 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).

1.4 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 the 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 malicious adversaries. However, one disadvantage of this approach is that the CRS is necessarily local, even in the semi honest case.

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 exploit an indirection technique similar to the one used in [KSW14, CPR16]: namely, 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 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. Note that the the only reason why the simulator needs to generate the CRS is a commitment scheme.

Third Attempt. So far our CRS is still local due to a commitment scheme. However, it turns out that we don't need the full power of the commitments; for the proof of security our commitment scheme should be statistically binding *only at round-1 commitments*, not everywhere. Since we are in the semi-honest setting, it is enough to have a commitment scheme that is statistically binding only on honestly generated commitments. 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: if we were using this scheme, the punctured program would depend 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 a symmetric key encryption scheme secure under simulation 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^*).

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

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. So far we described a protocol for deterministic functionalities. Here we describe how we handle randomized functionalities in a randomness-hiding way, i.e. the actual randomness used to compute the function should remain hidden even when all parties are corrupted and all their randomness is learned by the adversary.

It might seem first that to achieve randomness hiding we can use ideas of [SW14] and let the encryption program internally choose randomness by applying an extractor to the random input provided by a party - the technique used in both [CGP15,DKR14] to achieve randomness hiding. Namely, let the encryption program B generate a ciphertext containing not only input x_i of a party, but also randomness r_i derived internally by the program without help of the party. Later Eval can decrypt ciphertexts, learn all x_i and r_i and compute the function as $f(x_1, \ldots, x_n; \bigoplus r_i)$. However, this approach is bound to fail in our case: for our proof of security to go through, we crucially need the fact that *round-1 messages (i.e. commitments) completely determine the computation*, and therefore parties would have to commit to r_i in round 1. This means that parties have to know r_i themselves, and therefore the randomness of the computation will be revealed upon corruption.

Another idea to let our protocol compute randomized functionalities while hiding the randomness is to randomize program Eval in a natural way, i.e. let Eval apply a PRF on its inputs, and use the resulting randomness for computing the function. Hopefully, security of a PRF will guarantee that this randomness remains hidden. However, this idea still doesn't work in of itself: it again violates our crucial property that round-1 messages should determine the computation. Namely, if randomness was derived as a PRF of inputs to Eval (recall that Eval takes round-2 ciphertexts as inputs), this property would be violated, since for a given set of round-1 messages there may be many corresponding round-2 ciphertexts, and thus many possible randomness of the computation.

Our actual solution modifies the previous attempt so that the crucial computation-fixing property is not violated. For this, we let program Eval decrypt ciphertexts, compute a PRF *on round-1 commitments* and evaluate a randomized functionality with resulting 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.

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 $f(x_1, \dots, x_n; r)$, parties should use basic protocol to evaluate a randomized function $F(x_1, \dots, x_n; (r_1, r_2))$ which uses r_1 as randomness to garble function f and inputs $x_1, \dots, x_n, r_2$ (r_2 being random input of f).

1.5 Our Techniques: Malicious Case

To obtain a two-round RAM efficient protocol in a malicious setting, we observe that the protocol of [GP14] becomes RAM-efficient, as long as statistically-sound NIZK they use is RAM-efficient. Let us briefly describe their protocol. 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 and corresponding proofs. 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 $y = f(x_1, \dots, x_n)$, which is needed for the security proof to go through. As usual in "iO + NIZK" techniques, the NIZK has to be *statistically sound*. For all known NIZKs, this means that the verifier

[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).

(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, it suffices 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 circuit complexity of R (in particular, we need a garbling scheme where the size of circuits which generate garbling, i.e. the size of $\mathsf{GInp}, \mathsf{GProg}$, 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 observe that the garbling scheme of Canetti and Holmgren ([CH16]) already has both properties; see full version [CPV16] for details. Thus, our scheme yeilds a NIZK system when instantiated with the garbling scheme by [CH16].

Organization. Section 2 contains definitions and constructions of building blocks for our protocol, namely, of an honest-but-curious commitment and a puncturable randomized encryption. The protocol itself is given in Sect. 3, together with an overview of hybrids. The full proof of security and our NIZK is presented in the full version [CPV16]. The description of the malicious version of our main protocol is given in Appendix B.

[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.

2 Building Blocks

In this section we define and build *puncturable randomized encryption (PRE)* and *an honest-but-curious commitment* - primitives used in our MPC protocol (Sect. 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:*
  - *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*
  $\Pr[A(k\{c^*\}, m^*, c^*) = 1] - \Pr[A(k\{c_{\mathsf{Sim}}\}, m^*, c_{\mathsf{Sim}}) = 1] < \mathsf{negl}(\lambda)$.

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. Note that injective PRF doesn't require injective OWFs; instead, the existence of injective OFWs is required for the proof of [BST14] (that $\mathsf{iO}(\mathsf{PRF})$ is a hardcore function) to go through.

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

[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.

that $\mathsf{iO}(\mathsf{PRF})$ is a prg family, as long as there exist injective OWFs. 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 outputs $\perp$.
- **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.*

```
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.$K$, $\mathsf{PKE.pk}$)

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

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

– **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 Figs. 3 and 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.

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 $y \leftarrow \mathsf{Eval}(b_1, \ldots, b_n)$.

Fig. 2. MPC protocol.

In the first round everybody uses the semi-honest commitment scheme (defined and constructed in Sect. 2.2) to "commit" to (i, x_i) with randomness r_{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 central encryption scheme used by program B to encrypt and by Eval to decrypt is a puncturable randomized encryption (PRE), which we built in Sect. 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.*

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

[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 (Sect. 2.1) exists under this assumption and iO.

```
Programs in the CRS:

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

  – 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 $u_{\mathsf{Gen}}$ to sample extracting PRF key $\mathsf{Ext}_{\mathsf{B}}$, PRE key $K$, PRF key $k$, faking PDE key
         $f_{\mathsf{B}}$ 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 ExplainGen($\mathsf{Prog1}, \mathsf{Prog2}, \mathsf{Prog3}; \rho$)
Constants: faking PDE key $f_{\mathsf{Gen}}$

  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 $r_{\mathsf{Gen}}$.
```

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

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 functionalities, 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).

[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).

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 $\mathsf{Ext_B}$, faking PDE key $f_\mathsf{B}$, 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. $u_{B,i} \leftarrow F_{\mathsf{Ext_B}}(M, r_{B,i})$
      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 $f_\mathsf{B}$

  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 $r_{B,i}$
```

Fig. 4. Programs used in the protocol.

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 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).

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 $S \rightarrow S_i$. 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.

3.1 An Overview of the Hybrids

Here we present an overview of the hybrids. The full proof with security reductions is in the full version [CPV16].

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 (Sect. A).

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 (Sect. A).

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 $K\{\{b_i^*\}_{i=1}^n\}$ and a PRF key $k\{(a_1^*, \ldots, a_n^*)\}$ (see Fig. 5. 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 $K\{\{b_i^*\}_{i=1}^n\}$ 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

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 $\mathsf{Ext}_\mathsf{B}$, faking PDE key $f_\mathsf{B}$, 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 $y \leftarrow f(x_1, \ldots, x_n; R)$.
```

Fig. 5. Programs used in the proof and the simulation.

to the challenge set $a_1^*, \ldots, a_n^*$, then the program outputs hardwired y^*, if consistency checks pass. Recall that honestly generated $\{a_i^*\}_{i=1}^n$ 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 $K\{\{b_i^*\}_{i=1}^n\}$, PRF key $k\{(a_1^*, \ldots, a_n^*)\}$. Hardwired variables inside programs $\mathsf{B}, \mathsf{Eval}$ are $\{a_i^*\}_{i=1}^n$, $\{b_i^*\}_{i=1}^n, y^*$, 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 $\mathsf{Eval}, \mathsf{B}, \mathsf{ExplainB}$.

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.

A 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.

B Three Round MPC Against Malicious Adversaries

In this section we present our three-round, RAM-efficient, maliciously secure protocol with local CRS. Our protocol is described in Fig. 6. The CRS consists of two programs, Gen and $\mathsf{ExplainGen}$. The CRS will also contain a CRS σ_{CLOS} corresponding to the adaptively secure commitment scheme of [CLOS02] and a CRS σ_{NIZK} corresponding to a NIZK argument system that is simulation sound and secure against adaptive adversaries [GOS06].[16] We will denote

[16] We remark that the [GOS06] do not explicitly claim simulation soundness. It is easy to obtain a simulation-sound argument by sampling an independent CRS for every pair of parties.

by $\mathsf{adCom}_x(msg; r)$ the procedure to commit using the commitment scheme of [CLOS02] where x is the common reference string for the commitment, msg is the message and r is the randomness required. We will rely exactly on the same programs for Gen and $\mathsf{ExplainGen}$ from the semi-honest protocols described in Figs. 3 and 4. Recall that Gen is a generation algorithm which produces "encryption" program B, "decryption-and-evaluation" program Eval and program $\mathsf{ExplainB}$.

The protocol

CRS: σ_{CLOS}, σ_{NIZK} and programs Gen and $\mathsf{ExplainGen}$,
inputs: x_i; randomness: $r^1_{\mathsf{com},i}, r^2_{\mathsf{com},i}, r^3_{\mathsf{com},i}, \{r_{B,i,j}\}_{j=1,\ldots,n}, r_{\mathsf{Gen},i}$

1. **Round 1:** Each party P_i computes $a_i \leftarrow \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(i, x_i; r^1_{\mathsf{com},i})$, $\widetilde{r}_{\mathsf{Gen},i} \leftarrow \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(r_{\mathsf{Gen},i}; r^2_{\mathsf{com},i})$, $\widetilde{r}_{B,i,j} \leftarrow \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(r_{B,i,j}; r^3_{\mathsf{com},i})$ and broadcasts $(a_i, \widetilde{r}_{\mathsf{Gen},i}, \widetilde{r}_{B,i,j})$;
2. **Round 2:** Each party P_i broadcasts $r_{\mathsf{Gen},i}$, $\{r_{B,i,j}\}_{j \neq i}$ and proof Π_i of the statement S_i using an NIZK proof with CRS σ_{NIZK};
3. Each party sets $r_{\mathsf{Gen}} \leftarrow \bigoplus r_{\mathsf{Gen},i}$ and runs $\{\mathsf{B}, \mathsf{Eval}, \mathsf{ExplainB}\} \leftarrow \mathsf{Gen}(r_{\mathsf{Gen}})$;
4. **Round 3:** Each party broadcasts $b_i \leftarrow \mathsf{B}(i, x_i, r_{\mathsf{com},i}, a_1, \ldots, a_n; r_{B,i})$ where $r_{B,i} = \bigoplus_j r_{B,j,i}$;
5. Each party sets its output to be $y \leftarrow \mathsf{Eval}(b_1, \ldots, b_n)$.

Language S_i used in the protocol:

$S_i := ((\widetilde{r}_{\mathsf{Gen},i}, r_{\mathsf{Gen},i}, \widetilde{r}_{B,i,j}, r_{B,i,j}) : \exists r^2_{\mathsf{com},i}, r^3_{\mathsf{com},i},$ such that
$\widetilde{r}_{\mathsf{Gen},i} = \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(r_{\mathsf{Gen},i}; r^2_{\mathsf{com},i})$ and $\widetilde{r}_{B,i,j} = \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(r_{B,i,j}; r^3_{\mathsf{com},i}))$

Fig. 6. Malicious MPC protocol.

In the first round everybody uses the commitment scheme of [CLOS02] to separately commit to (i, x_i), $\{r_{B,i,j}\}_{j=1,\ldots,n}$ (to be used as a coin toss for encryption randomness) and $r_{\mathsf{Gen},i}$ (to be used as a coin toss for generation randomness).

In the second round, all parties reveal $r_{\mathsf{Gen},i}$ and $\{r_{B,i,j}\}_{j \neq i}$and prove using an NIZK proof that this is indeed the string committed to in the first round. More formally, party P_i proves the following NP-statement:

$$S_i := ((\ \widetilde{r}_{\mathsf{Gen},i}, r_{\mathsf{Gen},i}, \widetilde{r}_{B,i,j}, r_{B,i,j}) : \exists r^2_{\mathsf{com},\ i}, r^3_{\mathsf{com},\ i}, \text{ such that}$$
$$\widetilde{r}_{\mathsf{Gen},i} = \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(r_{\mathsf{Gen},i}; r^2_{\mathsf{com},\ i}) \text{ and } \widetilde{r}_{B,i,j} = \mathsf{adCom}_{\sigma_{\mathsf{CLOS}}}(r_{B,i,j}; r^3_{\mathsf{com},\ i})),$$

where $\widetilde{r}_{\mathsf{Gen},i}$ is defined in round 1 of the protocol and $r_{\mathsf{Gen},i}$ is the message revealed by party P_i in round 2. Then 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 the third round, all parties perform exactly the same instructions as they executed in round 2 of the semi-honest protocol. Namely, everybody runs the program B as: $b_i \leftarrow \mathsf{B}(i, x_i, r_{\mathsf{com},\ i}, a_1, \ldots, a_n; r_{B,i})$ (using randomness $r_{B,i} = \bigoplus_j r_{B,j,i}$) and broadcasts b_i. Then everybody computes $y \leftarrow \mathsf{Eval}(b_1, \ldots, b_n)$.

Theorem 7. *The protocol described above UC-securely implements $\mathcal{F}_{\mathsf{multi-f}}$ for any functionality f in the presence of malicious adaptive adversaries.*

We present a formal proof of the Theorem in the full version [CPV16].

References

[AIK06] Applebaum, B., Ishai, Y., Kushilevitz, E.: Computationally private randomizing polynomials and their applications. Comput. Complex. **15**(2), 115–162 (2006)

[BCH12] Bitansky, N., Canetti, R., Halevi, S.: Leakage-tolerant interactive protocols. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 266–284. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28914-9_15

[BCP15] Boyle, E., Chung, K.-M., Pass, R.: Large-scale secure computation: multi-party computation for (Parallel) RAM programs. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 742–762. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_36

[BST14] Bellare, M., Stepanovs, I., Tessaro, S.: Poly-many hardcore bits for any one-way function and a framework for differing-inputs obfuscation. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 102–121. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45608-8_6

[CDPW07] Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). doi:10.1007/978-3-540-70936-7_4

[CGP15] Canetti, R., Goldwasser, S., Poburinnaya, O.: Adaptively Secure Two-Party Computation from Indistinguishability Obfuscation. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 557–585. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46497-7_22

[CH16] Canetti, R., Holmgren, J.: Fully succinct garbled RAM. In: Proceedings of the ACM Conference on Innovations in Theoretical Computer Science. Cambridge, MA, USA, 14–16 January, pp. 169–178 (2016)

[CHJV15] Canetti, R., Holmgren, J., Jain, A., Vaikuntanathan, V.: Succinct garbling and indistinguishability obfuscation for RAM programs. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC. Portland, OR, USA, 14–17 June, pp. 429–437 (2015)

[CLOS02] Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In Proceedings on 34th Annual ACM Symposium on Theory of Computing, 19–21 May. Montréal, Québec, Canada, pp. 494–503 (2002)

[CPR16] Canetti, R., Poburinnaya, O., Raykova, M.: Optimal-rate non-committing encryption in a CRS model. IACR Cryptology ePrint Archive 2016:511 (2016)

[CPV16] Canetti, R., Poburinnaya, O., Venkitasubramaniam, M.: Better two-round adaptive multiparty computation. In: Cryptology ePrint Archive, Report 2016/614 (2016). http://eprint.iacr.org/2016/614

[DKR14] Dachman-Soled, D., Katz, J., Rao, V.: Adaptively secure, universally composable, multi-party computation in constant rounds. IACR Cryptology ePrint Archive 2014, 858 (2014)

[DMN11] Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 144–163. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19571-6_10

[Gen09] Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis. Stanford, CA, USA, AAI3382729 (2009)

[GGHR14] Garg, S., Gentry, C., Halevi, S., Raykova, M.: Two-round secure MPC from indistinguishability obfuscation. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 74–94. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54242-8_4

[GOS06] Groth, J., Ostrovsky, R., Sahai, A.: Perfect non-interactive zero knowledge for NP. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 339–358. Springer, Heidelberg (2006). doi:10.1007/11761679_21

[GP14] Garg, S., Polychroniadou, A.: Two-round adaptively secure MPC from indistinguishability obfuscation. IACR Cryptology ePrint Archive 2014:844 (2014)

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

[IK02] Ishai, Y., Kushilevitz, E.: Perfect constant-round secure computation via perfect randomizing polynomials. In: Widmayer, P., Eidenbenz, S., Triguero, F., Morales, R., Conejo, R., Hennessy, M. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 244–256. Springer, Heidelberg (2002). doi:10.1007/3-540-45465-9_22

[IKOS10] Ishai, Y., Kumarasubramanian, A., Orlandi, C., Sahai, A.: Proceedings on invertible sampling and adaptive security. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 466–482. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17373-8_27

[KSW14] Khurana, D., Sahai, A., Waters, B.: How to generate and use universal parameters. IACR Cryptology ePrint Archive 2014:507 (2014)

[NY90] Naor, M., Yung, M.: Public-key cryptosystems provably secure against chosen ciphertext attacks. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing. Baltimore, Maryland, USA, 13–17 May, pp. 427–437 (1990)

[SW14] Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: Symposium on Theory of Computing, STOC 2014, New York, NY, USA, 31 May-03 June, pp. 475–484 (2014)

[Wat15] Waters, B.: A punctured programming approach to adaptively secure functional encryption. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 678–697. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_33