Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption

Ivan Damgård and Jesper Buus Nielsen*

BRICS** Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Arhus C, Denmark

Abstract. We present a new general multiparty computation protocol for the cryptographic scenario which is universally composable — in particular, it is secure against an active and *adaptive* adversary, corrupting any minority of the parties. The protocol is as efficient as the best known statically secure solutions, in particular the number of bits broadcast (which dominates the complexity) is $\Omega(nk|C|)$, where n is the number of parties, k is a security parameter, and $|C|$ is the size of a circuit doing the desired computation. Unlike previous adaptively secure protocols for the cryptographic model, our protocol does not use non-committing encryption, instead it is based on homomorphic threshold encryption, in particular the Paillier cryptosystem.

1 Introduction

The problem of multiparty computation (MPC) dates back to the papers by Yao [14] and Goldreich et al. [11]. What was proved there was basically that a collection of n parties can efficiently compute the value of an n-input function, s.t. everyone learns the correct result, but no other new information. More precisely, these protocols can be proved secure against a polynomial time bounded adversary who can *corrupt* a set of less than $n/2$ parties initially, and then make them behave as he likes, we say that the adversary is *active*. Even so, the adversary should not be able to prevent the correct result from being computed and should learn nothing more than the result and the inputs of corrupted parties. Because the set of corrupted parties is fixed from the start, such an adversary is called *static* or non-adaptive.

Proving security of the protocol from [11] requires a complexity assumption, such as existence of trapdoor one-way permutations. This is because the model of communication considered there is s.t. the adversary may see every message sent

* `{ivan,buus}@brics.dk`.
** Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

between parties, this is sometimes known as the *cryptographic model*. Later, *unconditionally* secure MPC protocols were proposed by Ben-Or et al. and Chaum et al. [2, 6], in the model where *private* channels are assumed between every pair of parties. These protocols are secure, even against an *adaptive* adversary who may decide dynamically during the protocol who to corrupt.

Over the years, several protocols have been proposed which, under specific computational assumptions, improve the efficiency of general statically secure MPC, see for instance [10]. In particular, Cramer, Damgård and Nielsen [7] proposed a general MPC protocol that is secure against a static adversary corrupting any minority of the parties. The protocol assumes that keys for a threshold homomorphic cryptosystem have been set up, and has communication (broadcast) complexity $\Omega(nk|C|)$, where n is the number of parties, k is a security parameter, and $|C|$ is the size of a circuit computing the desired function. This is the so far the most efficient protocol known for the cryptographic model, tolerating a dishonest minority (which is the best possible if we want to guarantee termination). The homomorphic threshold cryptosystem can be the one of Paillier[13], or can be built from the quadratic residuosity assumption.

It is possible to get adaptive security, also in the cryptographic model: we can start from an adaptively secure protocol for the private channels model ([2, 6]), and then, instead of assuming perfect channels, we implement them using public-key encryption. While this will in general reduce adaptive security to static, using *non-committing encryption*, adaptive security carries over to the cryptographic model [4]. In [5], it is shown that non-committing encryption plus some additional techniques can provide MPC that is *universally composable*, an even stronger notion of security defined by Canetti [3]. All these protocols are, however, much less efficient than the best statically secure ones. An alternative approach that also gives adaptive security is to assume that parties can be trusted to securely erase certain critical data [1].

Our Results In this paper, we present a new general MPC protocol for the cryptographic model, based on Paillier encryption. The protocol is universally composable, in particular, it is secure against an active *adaptive* adversary corrupting any minority of the parties. Up to a constant factor, it is as efficient as the (statically secure) protocol from [7]. It is therefore the first general MPC solution for the cryptographic model where going to adaptive security does not cause a major loss in efficiency (or costs an extra assumption, such as secure erasures). It is also the first that does not use non-committing encryption, which may be of separate interest from a technical point of view. (The protocol in [5] is not for the private channels model, but it uses non-committing encryption for building adaptively secure oblivious transfer.) Instead of non-committing encryption we use some ideas from the universally composable commitments of [9], and combine this with the protocol from [7]. Thus we work on the same principle that parties supply encrypted input, and the protocol produces encryptions of the outputs, that the parties can then cooperate to decrypt. To make this secure, we introduce a new technique for randomizing encryptions before they are decrypted. This means that the adversary has no control over the encryptions that are decrypted, and this turns out to be essential for our proof of security to go through.

We prove our protocols secure in the framework for universally composable (UC) security by Canetti[3]. This framework allows to define the security of general reactive tasks, rather than just evaluation of functions. This allows us to prove that our protocol does not just provide secure function evaluation, but is in fact equivalent to what we call an *arithmetic black box* (ABB). An ABB can be thought of as a secure general-purpose computer. Every party can in private specify inputs to the ABB, and any majority of parties can ask it to perform any feasible computational task and make the result (and only the result) public. Moreover the ABB can be invoked several times and keeps an internal state between invocations. This point of view allows for easier and more modular proofs, and also makes it easier to use our protocols as tools in other constructions.

2 An Informal Description

In this section, we give a completely informal introduction to some main ideas. All the concepts introduced here will be treated more formally later in the paper. We will assume that from the start, the following scenario has been established: we have a semantically secure threshold public-key system given, i.e., there is a public encryption key *pk* known by all parties, while the matching private decryption key has been shared among the parties, s.t. each party holds a share of it.

We will use a threshold version of the Paillier cryptosystem, so the message space is $\mathbf{Z}_N$ for some RSA modulus N. For a plaintext $a \in \mathbf{Z}_N$, we let $\overline{a}$ denote an encryption of a. We have certain homomorphic properties: from encryptions $\overline{a}, \overline{b}$, anyone can easily compute (deterministically) an encryption of $a+b$, which we denote $\overline{a} \boxplus \overline{b}$. We also require that from an encryption $\overline{a}$ and a constant $\alpha \in \mathbf{Z}_N$, it is easy to compute a random encryption of αa, which we denote $\alpha \boxdot \overline{a}$. This immediately gives us an algorithm $\boxminus$ for subtracting.

We can then sketch how a computation was performed securely in the (statically secure) protocol from [7]: we assume the desired computation is specified as a circuit doing additions and multiplications in $\mathbf{Z}_N$ — This allows us to simulate a Boolean circuit in a straightforward way using 0/1 values in R. The protocol then starts by having each party publish encryptions of his input values and give zero-knowledge proofs that he knows these values. Then any operation involving addition or multiplication by constants can be performed with no interaction: if all parties know encryptions $\overline{a}, \overline{b}$ of input values to an addition gate, all parties can immediately compute an encryption of the output $\overline{a+b}$. This leaves only the problem of multiplications: Given encryptions $\overline{a}, \overline{b}$ (where it may be the case that no parties knows a nor b), compute securely an encryption of $c = ab$. This can be done by the following protocol (which is a slightly optimized version of the protocol from [7]):

1. Each party P_i chooses random $d_i \in \mathbf{Z}_N$ and broadcasts encryptions $\overline{d_i}$ and $\overline{d_i b}$.
2. All parties prove in zero-knowledge that they know their respective values of d_i, and that $\overline{d_i b}$ encrypts the correct value. Let S be the subset of the parties succeeding with both proofs.
3. All parties can now compute $\overline{a} \boxplus (\boxplus_{i \in S} \overline{d}_i)$. This ciphertext is decrypted using threshold decryption, so all parties learn $a + \sum_{i \in S} d_i$.
4. All parties set $\overline{c} = (a + \sum_{i \in S} d_i) \boxdot \overline{b} \boxminus (\boxplus_{i \in S} \overline{d_i b})$.

At the final stage we know encryptions of the output values, which we just decrypt using threshold decryption. Intuitively this is secure if the encryption is secure because, other than the outputs, only random values or values that should be known to the adversary are ever decrypted. This intuition was proved for a static adversary in [7]. But in the UC framework, where the adversary is adaptive, it is well known that there are several additional problems:

Loosely speaking, a proof of security requires building a simulator that simulates in an indistinguishable way the adversary's view of a real attack, while having access only to data the adversary *is supposed to know at any given time*. This means that for instance in the input stage the simulator needs to show the adversary encryptions that are claimed to contain the inputs of honest parties. At this time the simulator does not know these inputs, so it must encrypt some arbitrary values. This is fine for the time being, but if one of the honest parties are later corrupted, the simulator learns the real inputs of this party and must reveal them to the adversary along with a simulation of all internal data of the party. The simulator is now stuck, since the real inputs most likely are not consistent with the arbitrary values encrypted earlier.

We handle this problem using a combination of two tricks: first, we include in the public key an encryption $K = \overline{1}$. Then, we redefine the encryption method, and fix the rule that to encrypt a value a, one computes $a \boxdot K$. Under normal circumstances, this will be an encryption of a. The point, however, is that in the simulation, the simulator can decide what K should be, and will set $K = \overline{0}$. Then all encryptions used in the simulation will contain 0, in fact — using the algebraic properties of the encryption scheme — the simulator can compute C as an encryption of 0, store the random coins used for this and later make it seem as if C was computed as $C = a \boxdot K$ for any a it desires.

However, the simulator must also be able to find the input values the adversary supplies, immediately as the encryptions are made public. This is not possible if it only sees encryptions of 0. We therefore redefine the way inputs are supplied: for each input value x of party P_i, P_i uses the UC commitment scheme of [9] to make a commitment $\texttt{commit}(x)$ to x, and also sends an encryption $C = x \boxdot K$. Finally he proves in zero-knowledge that $\texttt{commit}(x)$ contains the same value as C. This can be done efficiently because [9] is also based on Paillier encryption, and the UC property of the commitments precisely allows the simulator to extract inputs of corrupted parties and fake it on behalf of honest parties.

A final problem we face is that the simulator will not be able to "cheat" in the threshold decryption protocol. The key setup for this protocol fixes the shares of the private key even in the simulation, so a ciphertext can only be decrypted to the value it actually contains. Of course, when decrypting the outputs, the correct results should be produced both in simulation and real life, and so we have a problem since we just said that all ciphertexts in the simulation really contain 0. We solve this by randomizing all ciphertexts before they are decrypted: we include another fixed encryption $R = \overline{0}$ in the public key. Then, given ciphertext C, the parties cooperate to create an encryption $r \boxdot R$, where r is a (secret) randomly chosen value that depends on input from all parties. Then we compute $C \boxplus (r \boxdot R)$ and decrypt this ciphertext. Under normal circumstances, it will of course contain the same plaintext as C. But in the simulation, the simulator will set $R = \overline{1}$, and "cheat" in the process where r is chosen, s.t. $r = a$, where a is the value the simulator wants the decryption to return. This works, since in the simulation any C actually encrypts 0.

3 Preliminaries

Notation Throughout the paper k will denote the security parameter. For a probabilistic polynomial time (PPT) algorithm A we will use $a \leftarrow A(x; r)$ to denote running A on input x and uniformly random bits $r \in \{0,1\}^{p(|x|)}$ producing output a; Here $p(\cdot)$ is a polynomial upper bounding the running time of A.

Non-erasure Σ-protocols Consider a binary relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ where $(x, w) \in R$ can be checked in PPT. By $L(R)$ we denote the set $\{x \in \{0,1\}^* | \exists w \in \{0,1\}^*((x, w) \in R)\}$.

A non-erasure Σ-protocol for relation R is six PPT algorithms $(A, l, Z, B, \mathtt{hvs}, \mathtt{rbs}, \mathtt{xtr})$ (and an integer l) specifying a three move, public randomness, honest verifier zero-knowledge protocol with special soundness. The prover has input $(x, w) \in R$ and the verifier has input x. The prover first computes a message $a \leftarrow A(x, w; r_A)$ and sends a to V. Then V returns a l-bit challenge e. The prover then computes a response to the challenge $z \leftarrow Z(x, w, r_a, e)$ and sends z to the verifier. The verifier then computes $b \leftarrow B(x, a, e, z)$, where $b \in \{0,1\}$ indicates whether to believe that the prover knows a valid witness w or not. The algorithm $\mathtt{hvs}$ is called the honest verifier simulator and takes as input $x \in L(R)$, $e \in \{0,1\}^l$ and a uniformly random bit-string $r_{\mathtt{hvs}}$ and produces as output (a, z) which is supposed to be distributed as the (a, z) produced by a honest prover with instance x receiving challenge e — this is defined formally below. The algorithm $\mathtt{rbs}$ is called the random bits simulator. It takes as input $(x, w) \in R$, a challenge $e \in \{0,1\}^l$ and bits $r_{\mathtt{hvs}}$, which we think of as the random bits used by $\mathtt{hvs}$ in a run $(a, z) \leftarrow \mathtt{hvs}(x, e; r_{\mathtt{hvs}})$, and it produces as output a bit-string r_A s.t. $a = A(x, w; r_A)$ and $z = Z(x, w, r_A, e)$. I.e. if (a, z) is the messages simulated using $\mathtt{hvs}$ given just $x \in L(R)$, then if w s.t. $(x, w) \in R$ later becomes known it is possible to construct random bits r_A s.t. it looks like (a, z) was generated as $a \leftarrow A(x, w; r_A)$ and $z \leftarrow Z(x, w, r_A, e)$. Finally $\mathtt{xtr}$ is a knowledge extractor, which given two correct conversations with the same first message can compute a witness. We now formalize these requirements along with completeness.

Completeness: For all $(x, w) \in R$, r_A and $e \in \{0,1\}^l$ we have that $B(x, A(x, w; r_a), e, Z(x, w, r_a, e)) = 1$.

Special non-erasure honest verifier zero-knowledge: The following two random variables are identically distributed for all $(x, w) \in R$ and $e \in \{0,1\}^l$:

$$\mathtt{EXEC}(x, w, e) = [a \leftarrow A(x, w; r_A); z \leftarrow Z(x, w, r_A, e) : (x, w, a, r_A, e, z)]$$
$$\mathtt{SIM}(x, w, e) = [(a, z) \leftarrow \mathtt{hvs}(x, e; r_{\mathtt{hvs}}); r_A \leftarrow \mathtt{rbs}(x, w, e, r_{\mathtt{hvs}}) : (x, w, a, r_A, e, z)]$$

Special soundness: For all $x \in S$ and (a, e, z) and (a, e', z') where $e \neq e'$, $B(x, a, e, z) = 1$ and $B(x, a, e', z') = 1$, we have that $(x, \mathtt{xtr}(x, a, e, z, e', z')) \in R$.

4 Non-Erasure Concurrent Zero-Knowledge Proofs of Knowledge

In [8] a concurrent zero-knowledge proofs of knowledge is presented based on any Σ-protocol. The protocol assumes that the prover P and verifier V agree on a key K for a trapdoor commitment scheme. The prover has input $(x, w) \in R$ and the verifier knows x. The protocol proceeds as follows.

1. The prover generates $a \leftarrow A(x, w; r_A)$, commits $c \leftarrow \mathtt{commit}_K(a; r_c)$ and sends c to the verifier.
2. The verifier sends uniformly random $e \in \{0,1\}^l$ to the prover.
3. The prover computes $z \leftarrow Z(x, w, r_A, e)$ and sends (a, r_c, z) to the verifier.
4. The verifier outputs 1 iff $c = \mathtt{commit}_K(a, r_c)$ and $B(x, a, e, z) = 1$.

To simulate the protocol without w one assumes that the simulator knows the trapdoor t of K. The simulator can then let c be a fake commitment and when it receives e compute $(a, z) \leftarrow \mathtt{hvs}(x, e; r_{\mathtt{hvs}})$, use t to compute r_c s.t. $c = \mathtt{commit}_K(a; r_c)$ and send (a, r_c, z). As observed in [9], if the Σ-protocol is non-erasure the above protocol is adaptively secure. Assume namely that P is corrupted and the simulator learns a witness. Then compute $r_A \leftarrow \mathtt{rbs}(x, w, e, r_{\mathtt{hvs}})$ and by special non-erasure honest verifier zero-knowledge the simulation is perfect. In the following we will call this to patch the state of the proof of knowledge.

In [8] a knowledge extractor $\mathtt{xtr}$ is given working as follows. Assume K is a random commitment key so that $\mathtt{commit}_K$ is computational binding and assume a corrupt prover succeeds in giving an acceptable proof with probability p say. Then by rewinding and giving new random challenges e' until a proof is accepted again we get, e.w.n.p., values a, e, z, a, e', z' s.t. $B(x, a, e, z) = B(x, a, e', z') = 1$ and $e \neq e'$ and can compute a witness w for x. The expected number of rechallenges needed is $\frac{1}{p}$ so if we run a proof and extract if a correct proof is given the expected number of rechallenges used in the extraction is $p\frac{1}{p}$ where p is the probability that the proof is accepted in the first place. This generalizes to several proofs run concurrently. Each time a proof is accepted invoke $\mathtt{xtr}$ and get a witness. If the context in which the proofs are run is PPT, then the running time, and the number of proofs, is bounded by a polynomial P, and so each invocation of `xtr` has expected running time less than P which with a total of at most P invocations, by the linearity of expectation, gives an expected polynomial running time of at most P^2.

Note that for the zero-knowledge simulator it is enough that K is a trapdoor commitment key, whereas for knowledge soundness it was needed that the key was random to ensure that $\texttt{commit}_K$ is computational binding. We can therefore let V pick the key K and send it to P. All we need to ensure is that is the simulator can get its hands on the trapdoor of K. We will use the protocol and simulator exactly this way later.

5 Universally Composable Security

In this section we give a sketch of the notion of universally composable security of synchronous protocols for the authenticated link model. Except for some minor technical differences it is the synchronous model described in [3].

A protocol π consists of n parties $P_1, \ldots, P_n$, all PPT interactive Turing machines (ITMs). The execution of a protocol takes place in the presence of an environment $\mathcal{Z}$, also a PPT ITM, which supplies inputs to and receives outputs from the parties. $\mathcal{Z}$ also models the adversary of the protocol, and so schedules the activation of the parties and corrupts parties. In each round r each party P_i sends a message $m_{i,j,r}$ to each party P_j; The message $m_{i,i,r}$ is the state of P_i after round r, and $m_{i,i,0} = (k, r_i)$ will be the security parameter and the random bits used by P_i. We model open channels by showing the messages $\{m_{i,j,r}\}_{j \in [n] \setminus \{i\}}$ to $\mathcal{Z}$, where $[n] = \{1, \ldots, n\}$. In each round $\mathcal{Z}$ inputs a value $x_{i,r}$ to P_i and receives and output $y_{i,r}$ from P_i. We write the r'th activation of P_i as $(\{m_{i,j,r}\}_{j \in [n]}, y_{i,r}) = P_i(\{m_{j,i,r-1}\}_{j \in [n]}, x_{i,r})$. We model that the parties cannot reliably erase their state by giving r_i to $\mathcal{Z}$ when P_i is corrupted. In the following C will denote the set of corrupted parties and $H = [n] \setminus C$. In detail the real-life execution proceeds as follows.

Init: The input is k, random bits $r = (r_1, \ldots, r_n) \in (\{0,1\}^*)^n$ and an auxiliary input $z \in \{0,1\}^*$. Set $r = 0$ and $C = \emptyset$. For $i, j \in [n]$ let $m_{i,j,0} = \epsilon$ for $i \neq j$ and $m_{i,i,0} = (k, r_i)$. Then input k and z to $\mathcal{Z}$ and activate $\mathcal{Z}$.

Environment activation: When activated $\mathcal{Z}$ outputs one of the following commands: ($\texttt{activate}\ i, x_{i,r}, \{m_{j,i,r-1}\}_{j \in C}$) for $i \in H$; ($\texttt{corrupt}\ i$) for $i \in H$; (`end round`); or ($\texttt{guess}\ b$) for $b \in \{0,1\}$. Commands are handled as described below and the environment is then activated again. We require that all honest parties are activated between two (`end round`) commands. When a ($\texttt{guess}\ b$) command is given the execution stops.

Party activation: $\{m_{j,i,r-1}\}_{j \in H}$ were defined in the previous round; Add these to $\{m_{j,i,r-1}\}_{j \in C}$ from the environment and compute $(\{m_{i,j,r}\}_{j \in [n]}, y_{i,r}) = P_i(\{m_{j,i,r-1}\}_{j \in [n]}, x_{i,r})$. Then give $\{m_{i,j,r}\}_{j \in [n] \setminus \{i\}}$ to $\mathcal{Z}$.

Corrupt: Give r_i to $\mathcal{Z}$. Set $C = C \cup \{i\}$.

End round: Give the value $\{y_{i,r}\}_{i \in H}$ defined in **Party activation** to $\mathcal{Z}$ and set $r = r + 1$.

The result of the execution is the bit b output by $\mathcal{Z}$. We denote this bit by $\texttt{REAL}_{\pi,\mathcal{Z}}(k,r,z)$. This defines a Boolean distribution ensemble $\texttt{REAL}_{\pi,\mathcal{Z}} = \{\texttt{REAL}_{\pi,\mathcal{Z}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$, where we take r to be uniformly random.

To define the security of a protocol an ideal functionality $\mathcal{F}$ is specified. The ideal functionality is a PPT ITM with n input tapes and n output tapes which we think of as being connected to n parties. The input-output behavior of the ideal functionality defines the desired input-output behavior of the protocol. To be able to specify protocols which are allowed to leak certain information $\mathcal{F}$ has a special output tape (SOT) on which it writes this information. The ideal functionality also has the special input tape (SIT). Each time $\mathcal{F}$ is given an input for P_i it writes some value on the SOT modeling the part of the input which is not required to be kept secret. When $\mathcal{F}$ receives the input ($\texttt{activate}$ v) a round is over, in response to which it writes a value on the output tape for each party. The value v models the inputs from the corrupted parties. When a party P_i is corrupted the ideal functionality receives the input ($\texttt{corrupt}$ i) on the SIT.

We then say that a protocol π realizes an ideal functionality $\mathcal{F}$ if there exists an interface, also called simulator, $\mathcal{S}$ which given access to $\mathcal{F}$ can simulate a run of π with the same input-output behavior. In doing this $\mathcal{S}$ is given the inputs of the corrupted parties, and the information leaked on the SOT of $\mathcal{F}$, and can specify the inputs of corrupted parties. In detail the ideal process, proceeds as follows.

Init: The input is k, random bits $r = (r_\mathcal{F}, r_\mathcal{S}) \in (\{0,1\}^*)^2$ and an auxiliary input $z \in \{0,1\}^*$. Set $r = 0$ and $C = \emptyset$. Provide $\mathcal{S}$ with $r_\mathcal{S}$, provide $\mathcal{F}$ with $r_\mathcal{F}$ and give k and z to $\mathcal{Z}$ and activate $\mathcal{Z}$.

Environment activation: $\mathcal{Z}$ is defined exactly as in the real-word, but now commands are handled by $\mathcal{S}$, as described below.

Party activation: Give $\{m_{j,i,r-1}\}_{i\in C}$ to $\mathcal{S}$ and give $x_{i,r}$ to $\mathcal{F}$ on the input tape for P_i and run $\mathcal{F}$ to get some value $v_\mathcal{F}$ on the SOT. This value is given to $\mathcal{S}$ which is then required to compute some value $\{m_{i,j,r}\}_{j\in[n]\setminus\{i\}}$ which is given to $\mathcal{Z}$.

Corrupt: When $\mathcal{Z}$ corrupts P_i, $\mathcal{S}$ is given the values $x_{i,0}, y_{i,0}, x_{i,1}, \ldots$ exchanged between $\mathcal{Z}$ and $\mathcal{F}$ for P_i. Furthermore ($\texttt{corrupt}$ i) is input on the SIT of $\mathcal{F}$ and $\mathcal{F}$ writes a value on the SOT; This value is given to $\mathcal{S}$. Then $\mathcal{S}$ outputs some value r_i which is given to $\mathcal{Z}$. Set $C = C \cup \{i\}$.

End round: $\mathcal{S}$ is activated and produces a value v. Then ($\texttt{activate}$ v) is input to $\mathcal{F}$ which produces output $\{y_{i,r}\}_{i\in[n]}$. Then $\{y_{i,r}\}_{i\in C}$ is given to $\mathcal{S}$ and $\{y_{i,r}\}_{i\in H}$ is given to $\mathcal{Z}$. Set $r = r + 1$.

The result of the ideal-process is the bit b output by $\mathcal{Z}$. We denote this bit by $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,r,z)$. This defines a Boolean distribution ensemble $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} = \{\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$.

Definition 1. *We say that π t-realizes $\mathcal{F}$ if there exists an interface $\mathcal{S}$ s.t. for all environments $\mathcal{Z}$ corrupting at most t parties it holds that $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \texttt{REAL}_{\pi,\mathcal{Z}}$.*

We then define the hybrid models. The $\mathcal{G}$-hybrid model is basically the real-life model where the parties in addition to the communication lines also have access to an ideal functionality $\mathcal{G}$. In each round all parties give an input to $\mathcal{G}$ and receive the output from $\mathcal{G}$ in the following round. The inputs to the SIT of $\mathcal{G}$ are given by $\mathcal{Z}$ and $\mathcal{Z}$ receives the outputs on the SOT. When defining security of a protocol in a hybrid model the interface $\mathcal{S}$ must in addition to the communication and internal state of parties also simulate the values exchanged between $\mathcal{G}$ and $\mathcal{Z}$. We call this a hybrid interface.

Definition 2. *We say that π t-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if there exists a hybrid interface $\mathcal{S}$ s.t. for all environments $\mathcal{Z}$ corrupting at most t parties it holds that $\mathtt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \mathtt{HYB}^{\mathcal{G}}_{\pi,\mathcal{Z}}$.*

A universal composability theorem can be proven for this framework. I.e. if a protocol realizes a given functionality it can be replaced for the functionality in all contexts. The above description easily generalizes to the case of several ideal functionalities in a hybrid model. In particular we will in the protocols following assume that the parties have access to an ideal functionality $\mathcal{F}_{\mathtt{BA}}$ for doing Byzantine Agreement. It expects a bit b_i as input from all parties. If all honest parties agree on a value v, then $\mathcal{F}_{\mathtt{BA}}$ outputs v to all parties. Otherwise the environment is allowed to determine the output through the SIT. We also assume a broadcast channel, which can easily be model with an ideal functionality.

6 The Paillier Cryptosystem and Some Tools

In this section we describe the Paillier cryptosystem [13]. The public key is a k-bit RSA modulus $pk = N = pq$, where p an q are chosen s.t. $p = 2p' + 1, q = 2q' + 1$ for primes p', q', and both p and q have $k/2$ bits. The plaintext space is $\mathbf{Z}_N$ and the ciphertext space is $\mathbf{Z}^*_{N^2}$. To encrypt $a \in \mathbf{Z}_N$, one chooses $r \in \mathbf{Z}^*_N$ uniformly at random and computes the ciphertext as $E_{pk}(a; r) = g^a r^N \bmod N^2$, where the element $g = N + 1$ has order N in $\mathbf{Z}^*_{N^2}$. The encryption function is homomorphic in the following sense $E_{pk}(a_1; r_1)E_{pk}(a_2; r_2) \bmod N^2 = E_{pk}(a_1 a_2 \bmod N; r_1 r_2 \bmod N)$ and $E_{pk}(a; r)^b = E_{pk}(ab \bmod N; r^b \bmod N)$. The private key is e.g. $sk = \phi(N)(\phi(N)^{-1} \bmod N)$, and it is straightforward to verify that $((N+1)^a r^N)^{sk} \bmod N^2 = Na+1$ from which $a \bmod N$ can be computed. Given a one can then compute r^N and $r = (r^N)^{N^{-1} \bmod \phi(N)} \bmod N^2$. Under an appropriate complexity assumption — the DCRA — this system is semantic secure. The DCRA states that random elements in $\mathbf{Z}^*_{N^2}$ are computationally indistinguishable from random elements of form $r^N \bmod N^2$.

For any ciphertext $K = E_{pk}(a; r_K)$ we can consider the function $E_{pk,K}(m; r) = K^m r^N \bmod N^2 = E_{pk}(am; s^m r \bmod N)$. If r is uniformly random in $\mathbf{Z}^*_N$, then $s^m r \bmod N$ is uniformly random in $\mathbf{Z}^*_N$ and so $E_{pk,K}(m; r)$ is a uniformly random encryption of $am \bmod N$. Notice that if $a \in \mathbf{Z}^*_N$, then from K and $c = E_{pk,K}(m; r)$ and sk we can efficiently compute m. Therefore $E_{pk,K}(m; r)$ is again a semantically secure encryption function. If on the other hand $a = 0$, then $E_{pk,K}(m; r)$ is a uniformly random encryption of 0. So, $E_{pk,K}(m; r)$ can be seen as a perfect hiding commitment scheme $\mathtt{commit}_{pk,K}(m; r) = E_{pk,K}(m; r)$. It is trivial to see that $\mathtt{commit}_{pk,K}(m;r)$ is computationally binding. If $K \in E_{pk}(1)$, then $\mathtt{commit}_{pk,K}(m;r)$ would be perfect binding, so no algorithm can find two different openings. An algorithm finding two different openings when $K = E_{pk}(0)$ would therefore distinguish encryptions of 1 from encryptions K of 0.

Notice furthermore that if $K = E_{pk}(0; r_K)$ and r_K is known, then given arbitrary $m, m' \in \mathbf{Z}_N$ and $r \in \mathbf{Z}_N^*$ we can compute $r' = s^{m-m'}r \bmod N$, so that $E_{pk,K}(m;r) = E_{pk,K}(m';r')$. All in all we have argued that if $K = E_{pk}(0; r_K)$ is a random encryption, then $\mathtt{commit}_K(m;r)$ is a perfect hiding computationally binding trapdoor commitment scheme with trapdoor t_K. When considering values $K \in \mathbf{Z}_{N^2}^*$ as keys we will call keys of the form $K = E_{pk}(0; t_K)$ trapdoor keys and we will call keys of the form $K = E_{pk}(a; s)$ for $a \in \mathbf{Z}_n^*$ encryption keys. We now describe a commitment scheme from [9].

A UC Commitment Scheme First we have to introduce a so-called double-trapdoor commitment scheme. Consider a trapdoor commitment scheme with key $K = (K_1, K_2)$, where K_1 and K_2 are both encryptions of 0. We commit as $\mathtt{commit}_{K_1,K_2}(m) = (E_{pk,K_1}(m_1), E_{pk,K_2}(m_2))$, where m_1 and m_2 are uniformly random values for which $m = m_1 + m_2 \bmod N$. To make a fake commitment just commit to random values m_1 and m_2, call the commitment (c_1, c_2). To trapdoor open such a commitment it is enough to know the trapdoor of one of K_1 and K_2. To open $c = (c_1, c_2)$ to m we can open c_1 to $m - m_2$ or we can open c_2 to $m - m_1$. We note for later use that the distribution of the random bits is independent of which trapdoor is used. Note that the domain of $\mathtt{commit}_K$ can be extended by committing block-wise. Here is the protocol from [9].

Setup: We assume that commitment sender S and receiver R agree on two independently chosen Paillier public-keys N and N' and a commitment key $K = (K_1, K_2) = (E_{N'}(0; t_1), E_{N'}(0, t_1)$. If R is honest at the beginning of the protocol we furthermore assume that t_1 and t_2 are uniformly random.

Commitment: S commits to $s \in \mathbf{Z}_N$ as follows.

1. S generate uniformly random $L_S \in \mathbf{Z}_{N^2}^*$, commits $c_L = \mathtt{commit}_{N',K}(L_S; r_L)$ and sends c_L to R.
2. R generates uniformly random $L_R \in \mathbf{Z}_{N^2}$ and sends L_R to R.
3. S computes $L = L_S L_R \bmod N^2$, commits to s under L as $c_s = \mathtt{commit}_L(s; r_s)$ for uniformly random $s \in \mathbf{Z}_N^*$ and sends c_s to R along with (L_S, r_L), and outputs (L, c_s, r_s).
4. R checks that $c_L = \mathtt{commit}_{N',K}(L_S; r_L)$ and if so computes $L = L_S L_R \bmod N^2$ and outputs (L, c_s).

In [9] a simulator $\mathcal{S}_{\mathtt{commit}}$ for the UC framework is given which has the property that given any so-called hitting commitment (L, c_s) it can simulate a run of the protocol which results in R outputting (L, c_s). All the simulator needs to do this is the trapdoor(s) of K when R is corrupted and an opening (s, r_s) of (L, c_s) when S is corrupted. The simulator has some properties given in Theorem 1.

Theorem 1. *Consider a simulation with hitting commitment (L, c_s) and output commitment (L', c'_s) from R.*

1. *If L is uniformly random from $\mathbf{Z}_{N^2}^*$ and c_s is a uniformly random commitment to s, then the simulation is distributed* exactly *as the real-life protocol on input s.*
2. *If S is honest after the simulation, then $(L', c_s') = (L, c_s)$.*
3. *If N' is a modulus for which $\texttt{commit}_{N',K}$ is computationally binding, then after a simulation where S was corrupted when c_L was sent and R was honest by the end of the simulation, L' is an encryption key, e.w.n.p.*

By simulating a trapdoor commitment under (pk, K) we mean the following. Generate $L = E_N(0; r_L)$ for uniformly random r_0 and compute $c_s = E_{N,L}(0; r_s)$ for uniformly random r_s. Then run $\mathcal{S}_{\texttt{commit}}$ with hitting commitment (L, c_s). We write the resulting commitment as $(L, c_s, ?)$. If at some point any $s \in \mathbf{Z}_N$ is given, then use r_0 and r_s to compute r_s' s.t. $c_s = E_{pk,N}(s; r_s')$ and give (s, r_s') to $\mathcal{S}_{\texttt{commit}}$ to patch the state of the simulation. We call this patching $(L, c_s, ?)$ to s.

A Non-Erasure Σ-Protocol In [9] a non-erasure Σ-protocol for identical plaintext is given for the Paillier cryptosystem. The instance is (K_1, c_1, K_2, c_2) and a witness is (s, r_1, r_2) s.t. $c_1 = E_{pk,K_1}(s; r_1)$ and $c_2 = E_{pk,K_2}(s; r_2)$.

Escape-Secure Threshold Decryption In this section we introduce the adaptively secure threshold decryption protocol from [12]. We first make a definition which will help us state the results precisely. We will introduce a weaker notion of UC security, which we call escape-security. An escape-simulator $\mathcal{S}$ is defined as a simulator in the UC framework augmented with a special escape state. An escape-simulation $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z)$ proceeds exactly as in the UC framework except that if $\mathcal{S}$ enters the escape state, then the simulation terminates with the output $\perp$ — we say that the simulation was escaped. For environment $\mathcal{Z}$ and values $k \in \mathbf{N}$ and $z \in \{0,1\}^*$ of the security parameter we define the simulation probability by $s_{\mathcal{Z}}(k, z) = 1 - \Pr[\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z) = \perp]$. When $s_{\mathcal{Z}}(k, z)$ is non-zero we define the conditional probabilities $c_{\mathcal{Z}}(k, z, b) = \frac{\Pr[\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)=b]}{s_{\mathcal{Z}}(k,z)}$. When $s_{\mathcal{Z}}(k, z)$ is zero we define $c_{\mathcal{Z}}(k, z, b) = \frac{1}{2}$ for $b \in \{0,1\}$. Then $(c_{\mathcal{Z}}(k, z, 0), c_{\mathcal{Z}}(k, z, 1))$ defines a Boolean distribution ensemble which we denote by $[\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} | \neg\perp]$.

Definition 3. *We say that π t-escape-securely realizes $\mathcal{F}$ if there exists an escape-simulator $\mathcal{S}$ s.t. $\texttt{REAL}_{\pi,\mathcal{Z}} \stackrel{c}{\approx} [\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} | \neg\perp]$ for all environments $\mathcal{Z}$ corrupting at most t parties.*

We now proceed to present the result from [12] in the above framework. Let (G, E, D) be a public-key cryptosystem and consider the following ideal functionality $\mathcal{F}_{(G,E,D)}$ for threshold decryption.

Init: In the first round it generates $(pk, sk) \leftarrow \mathcal{G}(k)$ and outputs pk to all parties and to the adversary.

Decryption: After a public key pk has been output we allow the parties to input a ciphertext C for decryption in any round and we also allow that more than one ciphertext is input in each round. Each ciphertext is handled as follows: If in some round r all honest parties input C, then send $(C, m = D_{sk}(C))$ to the adversary[1] and in round $r+6$ return m to all parties.

In our terminology here, what is proved in [12] is:

Theorem 2. *There exists a protocol $\pi_{\texttt{pal}}$ escape-securely realizing $\mathcal{F}_{\texttt{pal}}$ with simulation probability $\frac{1}{2}$.*

In [12] the key distribution is handled using a trusted party or a general MPC. Here we model this by assuming the key is generated by an ideal functionality $\mathcal{F}_{\texttt{pal},key-gen}$; It generates a random threshold key $(N, pv, sk_1, \ldots, sk_n)$ where N is a random Paillier public key, $sk_1, \ldots, sk_n$ are the private key shares of the parties and pv is public values used in the protocol for a.o.t. checking decryption shares. In [12] a decryption protocol $\pi_{\texttt{pal},dec}$ using this key is then described.

Their simulator consists of two parts. A simulator $\mathcal{S}_{\texttt{pal},key-gen}$ which given the public key pk from the ideal functionality simulates a key $(N, pv, sk_1, \ldots, sk_n) = \mathcal{S}_{\texttt{pal},key-gen}(pk)$ with the property that (N, pv) is computationally indistinguishable from a real key. Furthermore a simulator $\mathcal{S}_{\texttt{pal},dec}$ for the decryption protocol is given, which again is computationally indistinguishable from the real protocol as long as the simulation is not escaped. We need a last property of the protocol. Given a random Paillier key (pk, sk) it is possible to generate a threshold key $(pk, pv, sk_1, \ldots, sk_n)$ with the same distribution as $\mathcal{F}_{\texttt{pal},key-gen}$. We write $(pk, pv, sk_1, \ldots, sk_n) = \texttt{SingleToThesh}(pk, sk)$.

7 An Arithmetic Black Box

The ABB is an ideal functionality $\mathcal{F}_{\texttt{ABB}}$. Initially $\mathcal{F}_{\texttt{ABB}}$ outputs a uniformly random key $pk = N$ to all parties, defining the ring $\mathbf{Z}_N$ that $\mathcal{F}_{\texttt{ABB}}$ does arithmetic over. In each activation it expects a command from all honest parties and carries out the command if all honest parties agree. Typically agreement means that the parties gave the same command, e.g. $(x \leftarrow x_1 + x_2)$, but this does not always make sense. If e.g. party P_i is to load a secret value s into variable x, using the command $(P_i : x \leftarrow s)$, of course the other parties cannot acknowledge by giving the same command. In this case they input $(P_i : x \leftarrow ?)$; The intended meaning is that P_i is allowed to define x using a value unknown to the other parties. If two honest parties ever disagree $\mathcal{F}_{\texttt{ABB}}$ goes corrupted in the following sense: It outputs its entire current state and all future inputs on the SOT. Besides this it lets the environment determine all future outputs through the SIT. The functionality $\mathcal{F}_{\texttt{ABB}}$ holds a list of defined variable names x and stores a value $\texttt{val}(x) \in \mathbf{Z}_N$ for each. For all commands we require that variables on the right hand side of $\leftarrow$ are defined and that x is defined when a $(\texttt{output}\ x)$ command is given. In the command $(P_i : x \leftarrow x_1 \cdot x_2)$ we require that x_1 was defined by a $(P_i : x_1 \leftarrow ?)$ command. A violation of these requirements will make $\mathcal{F}_{\texttt{ABB}}$ go

[1] We return these values to the adversary to specify that it is not part of the functionality to keep C or m secret

corrupted. Each time a variable x is defined $\mathcal{F}_{\mathtt{ABB}}$ outputs $(\mathtt{defined}\ x)$ to all parties. Except for the s in $(P_i : x \leftarrow s)$ all input values are output on the SOT.

Init: Let c be the number of rounds used for setting up the keys in the real-life protocol. When the `init` command is given the ABB simply runs for c rounds ignoring all inputs. Then it generates a random Paillier key (N, sk) and outputs pk to all parties and outputs (N, sk) to the adversary.

Load: On $(P_i : x \leftarrow ?)$ in round r, if P_i inputs $(P_i : x \leftarrow s)$ for $s \in \mathbf{Z}_N$, then set $\mathtt{val}(x) = s$ in round $r+8$. If in a round before round $r+8$ P_i is corrupted and the adversary inputs $(\mathtt{change}\ s')$ on the SIT, then $\mathtt{val}(x) = s'$, and if the adversary inputs `fail`, then $\mathtt{val}(x)$ is not defined.

Linear combination: On $(x \leftarrow a_0 + \sum_{j=1}^{l} a_j x_j)$ for $a_j \in \mathbf{Z}_N$ define $\mathtt{val}(x) = a_0 + \sum_{j=1}^{l} a_j \mathtt{val}(x_j) \bmod N$ in the same round.

Private multiplication: On $(P_i : x \leftarrow x_1 \cdot x_2)$ in round r, if P_i also inputs $(P_i : x \leftarrow x_1 \cdot x_2)$, (this is an extra requirement as P_i might be corrupted), then in round $r+4$ define $\mathtt{val}(x) = \mathtt{val}(x_1)\mathtt{val}(x_2) \bmod N$. If in a round before round $r+5$ the adversary inputs `fail` on the SIT and P_i is corrupted, then $\mathtt{val}(x)$ is not defined.

Output: On $(\mathtt{output}\ x)$ in round r output $(\mathtt{output}\ x = \mathtt{val}(x))$ on the SIT and output $(\mathtt{output}\ x = \mathtt{val}(x))$ to all parties in round $r+12$.

The functionality has no general multiplication command, but running in the $\mathcal{F}_{\mathtt{ABB}}$-hybrid model, parties can do a multiplication of any two variable using the multiplication protocol from the introduction. We now describe a protocol $\pi_{\mathtt{ABB}}$ realizing $\mathcal{F}_{\mathtt{ABB}}$. A variable x with $\mathtt{val}(x) = s$ is represented by an encryption $\mathtt{enc}(x) = E_{pk,K}(s)$ on which the parties agree.

Init: The protocol runs in the hybrid model with access to two copies of the functionality $\mathcal{F}_{\mathtt{pal},key-gen}$ described in Section 6. We call the public key returned by the first copy $pk = N$, and we assume that the first copy also returns two random encryptions $K = E_{pk}(1)$ and $R = E_{pk}(0)$. We call the public key returned by the second copy $pk' = N'$, and we assume that the second copy also return $3n$ random encryptions $K_{i,l} = E_{pk'}(0)$ for $i \in [n], l \in \{0,1,2\}$. The key $K_{j,0}$ is for for running the proof of knowledge protocol from Section 4 with the relation in Section 6 with P_j as verifier. We will denote this by P_i proves to P_j. The double-trapdoor key $K_j = (N', K_{j,1}, K_{j,2})$ will be used for running the universally composable commitment scheme in Section 6 with P_j as receiver to generate commitments under N. When P_i is the committer with some value s we will denote this by P_i commits to s to P_j. In the first round the parties call the ideal functionalities and wait until all keys are returned.

Load: Below we use M as the key under which encryptions are made. When the parties carry out the load commands given from the environment they always use $M = K$. However the implementation of **Output** also uses the load command as an internal sub-routine, with $M = R$.

1. P_i computes $S = E_{pk,M}(s; r_S)$ and broadcasts S.
2. For $j \neq i$ party P_i commits to s to P_j. Denote the commitment by $(L_j, c_{s,j}, r_{s,j})$.
3. For $j \neq i$ party P_i proves to P_j with instance $x = (pk, M, S, L_j, c_{s,j})$ and witness $(s, r_S, s, r_{s,j})$.

4. For $j \neq i$ party P_j inputs 1 to $\mathcal{F}_{\texttt{BA}}$ iff the zero-knowledge proof given by P_i to P_j was accepted.
5. Wait until $\mathcal{F}_{\texttt{BA}}$ outputs a bit b. If $b = 1$ the parties set $\texttt{enc}(x) = S$.

Linear combination: Set $\texttt{enc}(x) = g^{a_0} \prod_{i=1}^{l} \texttt{enc}(x_j)^{a_j} \bmod N^2$.

Private multiplication: We assume that P_i knows (s, r_s) s.t. $\texttt{enc}(x_1) = E_{pk,K}(s; r_s)$.

1. P_i computes $X = E_{pk,\texttt{enc}(x_2)}(s; r_X)$ and broadcasts X.
2. For $j \neq i$ party P_i proves to P_j with instance $x = (pk, K, \texttt{enc}(x_1), \texttt{enc}(x_2), X)$ and witness (s, r_S, s, r_X), and as above the parties run a BA to determine whether to accept the proof.
3. If the result of the BA is 1 the parties set $\texttt{enc}(x) = X$.

Output:
1. P_i generates random $r_i \in \mathbf{Z}_N$ and loads it into variable x_i using $M = K$. Let $\{C_j\}_{j \neq i}$ denote the commitments used in Step 2 of the load.
2. Parti P_i loads r_i into variable y_i using $M = R$, but reuses the commitments $\{C_j\}_{j \neq i}$ from the previous load. If a party P_i input 0 to the BA in the previous load for P_j, then input 0 again in this load.
3. Let I be a size $t+1$ subset of the set of indices i for which y_i is now defined. Let $\{\lambda_i^I\}_{i \in I}$ be degree t Lagrange coefficients interpolating from $\{f(i)\}_{i \in I}$ to $f(0)$ over $\mathbf{Z}_N$. Compute $S = \prod_{i \in I} \texttt{enc}(y_i)^{\lambda_i^I} \bmod N^2$ and $T = \texttt{enc}(x)S \bmod N^2$. We assume some fixed way of picking the subset I so that all parties agree on T.
4. The parties run $\pi_{\texttt{pal},dec}$ from Section 6 on T and take as their output the value v returned by $\pi_{\texttt{pal},dec}$.

Theorem 3. *π_{ABB} $\lceil (n-1)/2 \rceil$-securely realizes $\mathcal{F}_{ABB}$.*

Due to space limitations we can only sketch the proof of Theorem 3, a full proof will appear in the Ph.D. dissertation of the second author. In the proof we construct an interface $\mathcal{S}$ simulating $\pi_{\texttt{ABB}}$ in the ideal process with access to $\mathcal{F}_{\texttt{ABB}}$. The simulator runs a copy of the protocol $\pi_{\texttt{ABB}}$ and keeps it consistent with the inputs and outputs of $\mathcal{F}_{\texttt{ABB}}$ in ideal process (in which $\mathcal{S}$ is run). In simulating $\pi_{\texttt{ABB}}$, $\mathcal{S}$ must provide inputs to $\mathcal{F}_{\texttt{ABB}}$ on behalf of the corrupted parties. For all commands except **Load** and **Private Multiplication** only the inputs from honest parties are consider by $\mathcal{F}_{\texttt{ABB}}$. However for a load $(P_i : x \leftarrow ?)$ and a private multiplication $(P_i : x \leftarrow x_1 \cdot x_2)$ the input from $\mathcal{S}$ matters. In the first case $\mathcal{S}$ must provide an input $(P_i : x \leftarrow s)$ for x to be defined and in the second case $\mathcal{S}$ must provide the input $(P_i : x \leftarrow x_1 \cdot x_2)$ for x to be defined. Whether the simulator supplies these inputs, and the value of s, depends on $\mathcal{Z}$: If $\mathcal{Z}$ lets P_i participate in the simulation of $\pi_{\texttt{ABB}}$ in a way which results in the honest parties defining $\texttt{enc}(x)$ to hold some encryption, then $\mathcal{S}$ will provide the input to $\mathcal{F}_{\texttt{ABB}}$ and in the case of a **Load** it will determine an appropriate value of s. The initialization is simulated as follows::

Init: For the first copy it receives the key $(pk = N, sk)$ from $\mathcal{F}_{\texttt{ABB}}$ and computes $(N, pv, sk_1, \ldots, sk_n) = \texttt{SingleToThesh}(pk, sk)$.(1) Then it computes $K = E_{pk}(0; r_K)$ and computes $R = E_{pk}(1; r_R)$ and outputs (pk, pv, sk_i, R, K) to party P_i. For the second copy it generates N' and all the keys $K_{j,l} = E_{N'}(0; r_{j,l})$ itself and distributes these as in the protocol. When running the simulator for the zero-knowledge proofs with P_j as verifier the key $K_{j,0}$ is given to the simulator, and if P_j is corrupted $t_{j,0}$

is given to the simulator. Notice that we do not use sk' so the commitment schemes $E_{pk',K_{j,0}}$ will be computationally binding as required. When running the simulator for a commitment from P_i to P_j the key $(K_{j,1}, K_{j,2})$ is given to the simulator, and if P_j is corrupted $t_{j,1}$ and $t_{j,2}$ are given to the simulator. Notice that except for K and R this initialization simulates perfectly the protocol.

The simulator maintains the invariant that a variable x is defined in the simulated $\pi_{\texttt{ABB}}$ iff it is defined in $\mathcal{F}_{\texttt{ABB}}$. Furthermore the simulator maintains that for all defined variables x it knows $\texttt{ran}(x)$ s.t. $\texttt{enc}(x) = E_{pk}(0; \texttt{ran}(x))$. These values are used for trapdoor openings under K, e.g. in the simulation of the load command:

Load: We only describe how to simulate for $M = K$ as for all loads with key R we will actually know the value to load, so the 'simulation' can be done by running the protocol. On $(P_i : x \leftarrow ?)$, if P_i is honest we proceed as follows:

1. P_i computes $S = E_{pk}(0; r_0)$ and broadcasts S. If P_i is corrupted after this step, then we learn $(P_i : x \leftarrow s)$ from the ideal process. Then using r_K and r_0 compute r_S s.t. $S = E_{pk,K}(s; r_S)$ and uses r_S as the internal state of P_i.
2. For $j \neq i$ party P_i simulates a trapdoor commitment to P_j giving commitment $(L_j, c_{s,j}, ?)$. If P_i is corrupted in or after this step, then first patch as as above, and then patch $(L_j, c_{s,j}, ?)$ to s.
3. For each party P_j where $j \neq i$ party P_i run the simulator from Section 4 with instance $x = (pk, K, S, (L_j, c_{s,j}))$. If P_i is corrupted after this Step, then first patch as above. Then $w = (s, r_S, r_{s,j})$ is a witness to x. Give this witness to the simulator from Section 4 to patch the state of the proof of knowledge.
4. The parties execute the BA following the protocol. If the result is 1 the parties set $\texttt{enc}(x) = S$ and $\texttt{ran}(x) = r_0$.

For corrupted P_i the simulator inputs $(P_i : x \leftarrow 0)$ to $\mathcal{F}_{\texttt{ABB}}$ on behalf of P_i and then lets the honest parties follow the protocol. If the BA fails the simulator inputs `fail` on the SIT of $\mathcal{F}_{\texttt{ABB}}$. If the broadcast value $\texttt{enc}(x) = S$ is accepted, then the simulator must at the end of the load input ($\texttt{change}$ s) on behalf of P_i on the SIT of $\mathcal{F}_{\texttt{ABB}}$. Since the BA output 1, at least $t+1$ parties input 1, so at least one party P_j which is still honest input 1. Let $(L_j, c_{s,j})$ be the commitment received by P_j. Since the proof received was accepted, P_i can, e.w.n.p., open $(L_j, c_{s,j})$ and S to the same value, s say — i.e. if we could rewind we could extract such a value. By Theorem 1 we can assume that if S was sent by the adversary, then L_j is a binding key, so (L_j, s_j) can only be opened to one value. Therefore we can using sk decrypt(2) $(L_j, c_{s,j})$ to obtain the value s to which P_i can open S. The intuition here is that if K had been an encryption of 1, then this value s, to which P_i can open S, would indeed be its plaintext value, so we have extracted the 'plaintext' of S.

If P_i is honest at the onset of the load but is corrupted before the third message of the commitment protocol is sent, i.e. in round $r + 2$, then the environment might send a commitment $(L_j, c_{s,j})$ to a value s' different from the s which has already been input to $\mathcal{F}_{\texttt{ABB}}$, and we have no guarantee that L_j is an encryption key. However, we have no need to decrypt either: Since the simulator sent S it patched the state of P_i to be consistent with $S = \texttt{enc}_{pk,K}(s; r_S)$. So, if K had been a binding key, then the state of P_i would be consistent with $\mathcal{F}_{\texttt{ABB}}$.

In all cases, if $\texttt{enc}(c)$ is accepted the simulator decrypts(3) $\texttt{enc}(x)$ to learn random bits $\texttt{ran}(x)$ s.t. $\texttt{enc}(x) = E_{pk}(0; \texttt{ran}(x))$.

A private multiplication is simulated similarly, letting $X = E_{pk}(0; r_0)$ and if s becomes known, computing r_X s.t. $X = E_{pk,\mathtt{enc}(x_2)}(s; r_X)$ — in doing this we use that we know $\mathtt{ran}(x_2)$ s.t. $\mathtt{enc}(x_2) = E_{pk}(0; \mathtt{ran}(x_2))$. Linear combination is simulated by letting $\mathtt{enc}(x) = g^{a^0} \prod_{i=1}^{l} \mathtt{enc}(x_j)^{a_j}$ and $\mathtt{ran}(x) = a_0 \prod_{i=1}^{l} \mathtt{ran}(x_j)^{a_j} \bmod N$. An output is simulated as follows:

Output: On input ($\mathtt{output}$ $x = v$) from $\mathcal{F}_{\mathtt{ABB}}$ we know that all honest parties got the input ($\mathtt{output}$ x) and will output v in 12 rounds. We cannot use the simulator from Section 6 to simulate a decryption to v, as the inconsistent party might get corrupted. Instead we cheating in the randomization as to make T an encryption of v, and then decrypt honestly.

1. For honest P_i run the load command by committing to uniformly random elements r'_i under trapdoor keys.
2. Let J be the indices j of corrupted parties for which the load into x_j succeeded. If $|J| < t$, then add the indices of some uniformly random honest parties until $|J| = t$ and for those parties, let $r_i = r'_i$. Pick a random degree $t+1$ polynomial $f(j)$ s.t. $f(j) = r_j$ for $j \in J$ and $f(0) = v$. Then for the honest parties set $r_i = f(i)$ and patch the load into x_i to be consistent with $s = r_i$. Then load r_i into y_i under R by running the load protocol honestly. This is possible as the previous load is now consistent with r_i.
3. This step is simulated by following the protocol. Since R is an encryption of 1, unless a corrupted party P_j was able to load different values into x_j and y_j, we will have that S is an encryption of $f(0) = v$ and so is then T.(4)
4. Run the decryption protocol as in the protocol.

We then prove $\mathtt{IDEAL}_{\mathcal{F}_{\mathtt{ABB}},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \mathtt{REAL}_{\pi_{\mathtt{ABB}},\mathcal{Z}}$ using a hybrids argument. Define $\mathbf{H'_0}$ by taking $\mathcal{F}_{\mathtt{ABB}}$, $\mathcal{S}$ and $\mathcal{Z}$ and running $\mathtt{IDEAL}_{\mathcal{F}_{\mathtt{ABB}},\mathcal{S},\mathcal{Z}}$ with the modification that $\mathcal{S}_{gen,\mathtt{pal}}(pk)$ is run at (*1*) to produce the key and $\mathcal{S}_{\mathtt{pal},dec}(T, D_{sk}(T))$ is run at (*4*) instead of $\pi_{\mathtt{pal},gen}$. Let $H_0 = [H_0|\neg\bot]$. Doing this we got rid of the use of sk at (*1*) but introduced one at (*4*). By the results in Section 6 we will have that $\mathtt{IDEAL}_{\mathcal{F}_{\mathtt{ABB}},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} H_0$. Define $\mathbf{H'_1}$ as H'_0 but at (*2, 3*) use $\mathtt{xtr}$ from Section 4 to extract the plaintext (and for (*3*), the random bits.) Let $H_1 = [H'_1|\neg\bot]$. By the results in Section 4 we will have $H_0 \stackrel{c}{\approx} H_1$. Define $\mathbf{H'_2}$ as H'_1 but at (*4*) run $\mathcal{S}_{\mathtt{pal},dec}(T, v)$ instead of $\mathcal{S}_{\mathtt{pal},dec}(T, D_{sk}(T))$. Let $H_2 = [H'_2|\neg\bot]$. By using Sections 4 it can be proved that $v = D_{sk}(T)$, e.w.n.p. So, $H_2 \stackrel{c}{\approx} H_1$. Notice that H_2 can be produced without using sk, so now we can use the semantic security of E_{pk}. Define $\mathbf{H'_3}$ as H'_2, but use $R = E_{pk}(0)$. It follows via semantic security that $H_3 = [H'_3|\neg\bot] \stackrel{c}{\approx} H_2$. Define $\mathbf{H'_4}$ as H'_3 except that we use correct inputs to all honest parties and pick the r_i values uniformly at random for all honest parties. It can be seen that $H_4 = [H'_4|\neg\bot] = H_3$, as $K, R \in E_{pk}(0)$, so no information is leaked about the inputs and the environment sees at most t of the r_i values and so cannot distinguish random values from random values consistent with a degree $t+1$ polynomial. Define $\mathbf{H'_5}$ as H'_4 but with $K = E_{pk}(1)$ and use semantic security to prove $H_5 = [H'_5|\neg\bot] \stackrel{c}{\approx} H_4$. Because $K = E(1)$ we cannot use the simulator from 6 for simulating the commitments anymore. However this not not needed either as from H'_4 all honest parties use correct inputs, so at the same time we start running all commitments honestly. Define $\mathbf{H_6'}$ as H_5' but starting to use sk at (*2*) again and as for $H_0 \stackrel{c}{\approx} H_1$ get that $H_6 = [H_6'|\neg\bot] \stackrel{c}{\approx} H_5$. Define $\mathbf{H_7'}$ as H_6' but at (*4*) run $\mathcal{S}_{\mathtt{pal},dec}$ on $(T, D_{sk}(T))$ instead. Proving $H_7 = [H_7'|\neg\bot] \stackrel{c}{\approx} H_6'$ basically involves proving the protocol correct, which is done using the results from Sections 4. The only difference between H_7 and $\mathtt{REAL}_{\pi_{\mathtt{ABB}},\mathcal{Z}}$ is between the use of $\mathcal{S}_{\mathtt{pal},key-gen}$ and $\mathcal{S}_{\mathtt{pal},dec}$ instead of $\mathcal{F}_{\mathtt{pal},key-gen}$ and $\pi_{\mathtt{pal},dec}$, and $H_7 \stackrel{c}{\approx} \mathtt{REAL}_{\pi_{\mathtt{ABB}},\mathcal{F}}$ follows as $\mathtt{IDEAL}_{\mathcal{F}_{\mathtt{ABB}},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} H_0$.

References

1. D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In R. A. Rueppel, editor, *Advances in Cryptology - EuroCrypt '92*, pp. 307–323, Berlin, 1992. Springer-Verlag. LNCS Vol. 658.
2. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th STOC*, pp. 1–10, Chicago, Illinois, May 1988.
3. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42th FOCS*. IEEE, 2001.
4. R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *28th STOC*, pp. 639–648, Philadelphia, Pennsylvania, May 1996.
5. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *34th STOC*, pp. 494–503, Montreal, Quebec, Canada, 2002.
6. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th STOC*, pp. 11–19, Chicago, Illinois, May 1988.
7. R. Cramer, I. Damgaard, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - EuroCrypt 2001*, pp. 280–300, Berlin, 2001. Springer-Verlag. LNCS Vol. 2045.
8. I. Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In B. Preneel, editor, *Advances in Cryptology - EuroCrypt 2000*, pp. 418–430, Berlin, 2000. Springer-Verlag. LNCS Vol. 1807.
9. I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In M. Yung, editor, *Advances in Cryptology - Crypto 2002*, pp. 581–596, Berlin, 2002. Springer-Verlag. LNCS Vol. 2442.
10. R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multi-party computations with applications to threshold cryptography. In *PODC'98*, 1998.
11. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *19th STOC*, pp. 218–229, New York City, May 1987.
12. A. Lysyanskaya and C. Peikert. Adaptive security in the threshold setting: From cryptosystems to signature schemes. In C. Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001*, pp. 331–350, Berlin, 2001. Springer. LNCS Vol. 2248.
13. P. Paillier. Public-key cryptosystems based on composite degree residue classes. In J. Stern, editor, *Advances in Cryptology - EuroCrypt '99*, pp. 223–238, Berlin, 1999. Springer-Verlag. LNCS Vol. 1592.
14. A. C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*. IEEE. 1982. pp. 160–164, Chicago, Illinois, 3–5 Nov. 1982.