On the Theoretical Gap Between Synchronous and Asynchronous MPC Protocols*

Zuzana Beerliová-Trubíniová
ETH Zurich, Switzerland
bzuzana@inf.ethz.ch

Martin Hirt
ETH Zurich, Switzerland
hirt@inf.ethz.ch

Jesper Buus Nielsen
University of Aarhus, Denmark
jbn@cs.au.dk

ABSTRACT

Multiparty computation (MPC) protocols among n parties secure against t active faults are known to exist if and only if

- $t < n/\mathbf{2}$, when the channels are *synchronous*, and
- $t < n/\mathbf{3}$, when the channels are *asynchronous*, respectively.

In this work we analyze the gap between these bounds, and show that in the cryptographic setting (with setup), the sole reason for it is the *distribution of inputs*: given an oracle for input distribution, cryptographically-secure asynchronous MPC is possible with the very same condition as synchronous MPC, namely $t < n/2$. We do not know whether the gaps in other security models (perfect, statistical) have the same cause. We stress that all previous asynchronous MPC protocols inherently require $t < n/3$, even once inputs are distributed. In particular, all published asynchronous multiplication sub-protocols inherently require $t < n/3$ and cannot be used in our setting.

Furthermore, we show that such an input-distribution oracle can be reduced to an oracle that allows each party to synchronously broadcast one single message. This means that when one single round of synchronous broadcast is available, then asynchronous MPC is possible at the same condition as synchronous MPC, namely $t < n/2$. If such a round cannot be used, then MPC (and even Byzantine agreement) requires $t < n/3$.

Categories and Subject Descriptors

F.0 [**Theory of Computation**]: General

General Terms

Theory

Keywords

Cryptography, Asynchronous network, Multi-party computation, MPC

*This work was supported by the Zurich Information Security Center, and by the Danish Agency for Science Technology and Innovation. It represents the views of the authors.

PODC'10, July 25–28, 2010, Zurich, Switzerland.
Copyright 2010 ACM 978-1-60558-888-9/10/07 ...$10.00.

1. INTRODUCTION

1.1 Multiparty Computation

Secure multiparty computation (MPC) allows a set of n parties to securely evaluate any agreed function of their inputs, even if t of the parties are corrupted by a central adversary. In this paper we focus on an active adversary, that can take full control over the corrupted parties (i.e., read their internal state and make them send wrong messages). A protocol is called secure if the uncorrupted parties output the correct function values (*correctness*), and if the adversary does not learn anything that cannot efficiently be derived from the inputs and outputs of the corrupted parties (*privacy*).

The MPC problem dates back to Yao [Yao82], and the first generic solutions were presented in [GMW87, CDG87]. These protocols are secure for $t < n/2$, and this is known to be optimal.[1]

1.2 Synchronous vs. Asynchronous Communication

In the literature, mainly two communication models are considered: In the *synchronous model*, it is assumed that the delay of messages in the network is bounded by a *known* constant. This allows protocols to proceed in rounds, with the guarantee that every message sent in some round will be delivered at the beginning of the next round. In contrast, in the *asynchronous model*, arbitrary delays in the network are allowed, with the only restriction that every sent message must eventually be delivered. In order to model the worst case, the adversary is allowed to control the scheduling of messages in the network.

Summarizing, the synchronous model is very convenient, but not at all realistic, and the asynchronous model is quite realistic, but much less convenient. In fact, the bounds for the feasibility of MPC differ depending on whether the underlying network is synchronous (then $t < n/2$ is possible) or asynchronous (then $t < n/3$ is required).

1.3 Contributions

In this paper, we analyze the theoretical gap in the conditions for synchronous MPC (i.e., $t < n/2$) and asynchronous MPC (i.e., $t < n/3$), and demonstrate that asynchronous MPC is possible for $t < n/2$ if an "input distribution oracle" is available (respectively, if the inputs are predistributed). Hence, the reason why $t < n/3$ is required

[1]For $t < n$, still some reduced notion of security is achievable.

in asynchronous MPC is the input phase. We stress that all previous asynchronous MPC protocols [BCG93, BKR94, HNP05, BH07] cannot handle $t < n/2$, even when an appropriate input distribution oracle is available: All these protocols proceed by evaluating the circuit gate-by-gate, incrementally obtaining a sharing or a probabilistic encryption of each wire. Obviously, obtaining a *consistent* sharing or encryption of a wire immediately implies Byzantine agreement, which provably is not possible in an asynchronous network with $t \geq n/3$ [Tou84].

Furthermore, we show that the input distribution oracle can be reduced to (a single invocation of) an oracle that allows each party to synchronously broadcast one single message. This means that when one single round of synchronous broadcast is available, then asynchronous MPC is possible at the same condition as synchronous MPC, namely $t < n/2$. This is the weakest synchronicity assumption for which MPC with $t < n/2$ is known to exist (c.f. [FN09]).

The resulting MPC protocol provides cryptographic security against a static, computationally bounded adversary corrupting $t < n/2$ parties. The protocol even provides input guarantee, i.e., it allows every player to provide input.[2] However, the protocol requires quite a strong setup, similar to [CDN01, HNP05]. We do not know whether $t < n/2$ can be achieved with weaker or even without setup assumptions.

Several of our new techniques are also of independent interest: For reducing the input distribution oracle to the broadcast oracle (with $t < n/2$), we present an almost non-interactive verifiable secret-sharing scheme, and an almost non-interactive zero-knowledge proof of knowledge. Furthermore, we present the first asynchronous multiplication protocol for $t < n/2$. We stress that all previously known asynchronous multiplication protocols inherently require $t < n/3$, and cannot easily be pimped to $t < n/2$ (our focus). In other words, no asynchronous multiplication protocol in the literature can be applied to our setting with $t < n/2$. As a consequence, our multiplication protocol is different from all protocols in the literature, see Section 4.1 for details.

2. PRELIMINARIES

2.1 Model

We consider a set of n parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, each P_i holding an input x_i. The faultiness of parties is modeled by a central poly-time adversary who can corrupt up to $t < n/2$ of the parties (for a given threshold t) and make them deviate from the protocol in any desired matter. The number of actually corrupted parties is denoted by f.

The parties are connected by a network of asynchronous, authenticated point-to-point channels. The messages can be delayed arbitrarily and the order of the messages does not have to be preserved (however every sent message is eventually delivered). The computation proceeds in steps. In every step one party is active—it is activated by receiving a message, then it performs some local computation and eventually sends out some messages. To model the worst case scenario, we give the power to schedule the message delivery to the adversary—he can choose in every step which of the messages in the network is to be delivered.

Our protocol can be proved statically secure in a simulation-based sense [Can00]. We conjecture that using the techniques that were used by [DN03] in order to present an adaptively secure version of [CDN01], our protocol can be modified to be adaptively secure as well. However, since adaptive security is not the focus of this paper, the considerable complications needed to obtain adaptive security would only blur the focus of the paper.

We assume a trusted setup allowing threshold signatures and encryption and concurrent non-malleable zero-knowledge proofs. Details are given in the following subsections.

2.2 Threshold Signatures

We use a threshold signature scheme with threshold t. There is a publicly known *verification key* vk and a secret *signature key* sk. Each P_i holds a *signature key share* sk_i. Given a message m the party P_i can compute a *signature share* σ_i on m and can prove to any other party, using a two-party zero-knowledge protocol, that σ_i is a correct signature share on m. Given the verification key vk and $t + 1$ correct signature shares, anyone can compute a signature $\sigma = \sigma_{sk}(m)$. The system is unforgeable by a poly-time adversary knowing up to t of the shares sk_i—it takes a signature share from at least one honest party to create a valid signature on m under vk. The system from [Sho00] meets these requirements.[3]

2.3 Threshold Homomorphic Encryption

Our protocols will use ideas from [HNP05], which uses threshold homomorphic encryption to implement asynchronous MPC. One possible instantiation of threshold homomorphic encryption is using Paillier's encryption system [Pai99] as used in [CDN01]. The details are described in [HNP05], but all we need for the level of discussion in this extended abstract is the following.

Threshold Decryption.

In a threshold system, with threshold t, there is an *encryption key* ek and a *decryption key* dk. The encryption key is known by all parties and dk is shared among the parties, with each P_i holding a share dk_i. Given ek, a *plaintext* x and a *randomizer* r anyone can compute a *ciphertext* $X = E_{ek}(x; r)$. For a ciphertext X each P_i can compute a *decryption share* X_i and can prove, using a two-party zero-knowledge protocol, that X_i is a correct decryption share. Given the encryption key ek and $t + 1$ correct decryption shares, from different parties, anyone can compute the plaintext $x = D_{dk}(X)$. The system is indistinguishable under chosen plaintext attack (IND-CPA) against an adversary knowing t of the shares dk_i.

If such a system has been setup and $t < n/2$, then any party P_d, which is allowed to, can decrypt a ciphertext X asynchronously: The party sends X to all parties. Each party which agrees that P_d is allowed to decrypt X sends a decryption share to P_d and proves to P_d that the decryption share is correct. The party P_d waits for $n-t$ shares to arrive for which valid proofs were provided. Since there are $n - t$

[2] Normally, asynchronous MPC protocols ignore the inputs of up to t honest parties. This can only be prevented with additional synchronicity assumptions and special technical tricks [HNP05, BH07].

[3] The system from [Sho00] uses the random oracle model to be non-interactive. To avoid the random oracle we simply use interactive proofs, as described in e.g. [Nie02].

honest parties, if P_d is allowed to decrypt X, then this is deadlock free. And, since $n - t \geq t + 1$, P_d eventually gets enough shares to compute $x = D(X)$.

Homomorphic Encryption.

We assume that the encryption is homomorphic modulo some publicly known integer N. There exists some operation $\boxplus$ on ciphertexts such that $E_{ek}(x; r) \boxplus E_{ek}(y; s) = E_{ek}(x + y \bmod N; t)$ for some randomizer t, which can be computed efficiently from x, r, y, s if these are known. We also assume that given $X = E_{ek}(x; r)$ one can compute $X' = E_{ek}(N - x; s)$ for some randomizer s, which can be computed efficiently from x and r. We assume that these operations can be performed efficiently given just the encryption key ek. We use $C \in E(x)$ to mean that there exists a randomizer r such that $C = E_{ek}(x; r)$. Note that by combining the homomorphic properties one can take any ciphertext $B \in E(b)$ and any integer $a \in \mathbb{Z}_N$ and efficiently compute an encryption $C \in E(ab \bmod N)$ using double-and-add. We call this *multiplication by a constant* and write $C = a \boxdot B$. We write $A \boxminus B$ for $A \boxplus (-1 \boxdot B)$. We also assume that it is possible to take any ciphertext $C \in E(c)$ and efficiently compute a *uniformly random* ciphertext $C' \in_R E(c)$, using just the encryption key. We write $C' \leftarrow [C]$ and call this *re-randomization*. We write $C' = [C](r)$ when we want to make explicit the randomness r used for re-randomization. To guarantee robustness of some of our protocols we assume that there exists a concurrent, non-malleable zero-knowledge (ZK) proof which allows a party having computed $C' = [a \boxdot B](r)$ to prove to another party that it knows a such that there exists r such that $C' = [a \boxdot B](r)$—the verifier is expected to know just ek, B and C'.

2.4 Concurrent, Non-malleable ZK

All ZK proofs mentioned above, and in the following sections, can be implemented as in [CDN01, HNP05] by transforming three-move, public-randomness, honest-verifier ZK proofs as described in [Dam00]. This yields concurrent, non-malleable ZK proofs for the common reference string model.

3. PROTOCOL OVERVIEW

On a high level, our protocol follows the standard approach with homomorphic threshold encryption, along the lines of [FH96, CDN01, HNP05]. At the beginning, an encryption of each party's input is distributed. Then, the agreed function is evaluated gate-by-gate, where for each gate, an encryption of its value is computed. Finally, the value of the output gate(s) is decrypted using threshold decryption.

As a matter of fact, the above description is not quite true: In an asynchronous model with $t < n/2$, no agreement on whatsoever can be achieved (provably, with $t \geq n/3$ Byzantine agreement is impossible[Tou84]). Hence, the parties cannot reach agreement, neither on their input values, nor on the (probabilistic) encryptions of intermediate values. The latter issue is avoided with a technical trick: in our protocol, the players do not reach agreement on encryptions, but only on the plaintexts inside the encryptions. I.e., of the very same value, several different encryption are flowing around. The first issue (agreement on inputs) provably cannot be avoided in an asynchronous network with $t < n/2$ (this would imply Byzantine agreement). Therefore, we simply assume an input distribution oracle (which later will be reduced to a broadcast oracle).

The impossibility of Byzantine agreement in our model implies that the function to be evaluated must be *deterministic*. However, probabilistic polynomial-time (PPT) functions can easily be computed by evaluating a deterministic function on the actual inputs and some additional random inputs provided by the parties. Furthermore, for the sake of simplicity we assume that the function has public outputs only. Also this restriction can easily be overcome by letting the parties input random pads that are XORed on their local outputs. Thus we can assume without loss of generality that the function to be computed is deterministic with public outputs.

We prove the following results.

THEOREM 1. *For $t < n/2$, any PPT function can be evaluated on predistributed inputs over an asynchronous network.*

THEOREM 2. *For $t < n/2$, any PPT function can be computed over an asynchronous network, when one synchronous broadcast round is available.*

In the following section, we describe the asynchronous MPC protocol with predistributed inputs. In the subsequent section, we describe the input stage when given a single round of synchronous broadcast. Finally, we discuss under which assumptions this broadcast round can be simulated.

4. ASYNCHRONOUS MPC WITH $t < n/2$ AND PREDISTRIBUTED INPUTS

In this section we present an asynchronous MPC protocol which allows to distributively evaluate an agreed function on *predistributed inputs*. This protocol tolerates $t < n/2$ corrupted parties, which means (among other things), that Byzantine agreement cannot be achieved.

The function is evaluated in the usual gate-by-gate manner. Starting with the given input encryptions, the parties jointly compute encryptions of each intermediary value (one after the other), until eventually an encryption of the output is available and jointly decrypted (using threshold decryption). As asynchronous Byzantine agreement is not possible for $t < n/2$, agreement on the *encryptions* of intermediary values cannot be guaranteed (however, agreement on the intermediary values is possible, as they can be deterministically derived from the predistributed inputs).

We solve the issue of inconsistent views on encryptions by evaluating the whole circuit many times in parallel, once for every party, denoted as *king*. The other parties act as *slaves* and help the king evaluating his copy of the circuit. When the king is honest, then all slaves will have consistent views on all encryptions. When the king is faulty, inconsistencies will occur, but we will show that they do not violate privacy (by cheating, the king learns either the correct output or some uniformly random value).

The protocol proceeds in two phases: In the *computation phase*, the circuit is evaluated n times in parallel, once for every king. In the subsequent *termination phase*, the parties ensure that all parties have learned the output, and hence all programs can safely be stopped. Note that not necessarily all kings can (or must) finish their copy of the circuit; once

$t+1$ kings have finished *with the same output*, then obviously this must be the correct output, and all parties adopt this value and stop.

4.1 Computation Phase

We assume that for every input wire, the parties have agreement on the ciphertext $X = E(x)$ of the input value x. Then every king P_k runs (with the help of the other parties acting as slaves) his own circuit evaluation, learning an encryption of the output of every gate. Throughout the whole computation it holds: whenever an honest party holds a ciphertext X for the output wire of some gate, then indeed $x = D(X)$ is the correct value of that wire;[4] however, we do not require that all slaves hold the same encryption X of a wire when the king is faulty. To every gate a unique *gate id* *gid* is assigned. In the following, we present the protocols for addition, output, and multiplication gates.

4.1.1 Addition Gate:

Whenever a slave P_i of P_k holds ciphertexts X and Y of the input wires of an addition gate *gid*, he computes $Z = X \boxplus Y$ as encryption of the output wire.

4.1.2 Output Gates:

Whenever a slave P_i of P_k holds a ciphertext Z of an output gate *gid*, he sends to P_k a decryption share of Z, and gives a(n interactive) ZK proof that the decryption share is correct for Z. Once P_k holds a ciphertext Z of the output gate *gid*, and receives $t+1$ valid decryption shares for this Z, he computes the output z for gate *gid*.

4.1.3 Multiplication Gates:

For multiplication, first the slaves help the king to generate a random multiplication triple [Bea91]. This triple consists of two encrypted random factors and the corresponding encrypted product. The actual multiplication is then evaluated with help of this prepared triple.

Intuitively, the generation of the multiplication triple proceeds as follows: P_k starts with the *initial triple* $(A_0, B_0, C_0) = \big(E(1;\epsilon), E(1;\epsilon), E(1;\epsilon)\big)$, where ϵ denotes some fixed agreed-upon randomness for encryption. Trivially, (A_0, B_0, C_0) is a correct multiplication triple (though far from being random). Then, in turn for $j = 0, \ldots, t$, P_k sends (A_j, B_j, C_j) to some party, who randomizes it to $(A_{j+1}, B_{j+1}, C_{j+1}) = \big(A_j \boxplus E(u), B_j \boxplus E(v), C_j \boxplus E(uv) \boxplus (u \boxdot B_j) \boxplus (v \boxdot A_j)\big)$ for randomly chosen $u, v \in \mathbb{Z}_N$, and sends back to P_k the new triple $(A_{j+1}, B_{j+1}, C_{j+1})$ along with a ZK proof that it was correctly generated. Clearly, $(A_{t+1}, B_{t+1}, C_{t+1})$ is still a correct multiplication triple. Furthermore, as $t+1$ parties have randomized the triple, at least one of them being honest, the resulting triple is a random multiplication triple.

We first present the protocol that allows a party P_i to randomize a triple (A_j, B_j, C_j) to $(A_{j+1}, B_{j+1}, C_{j+1})$, and get the new triple certified to be a correct j-th randomization for gate *gid* by party P_i for king P_k.

Protocol RandomizeTriple.

0. P_i has input P_k, *gid*, j, and (A_j, B_j, C_j).
1. P_i picks uniformly random plaintexts $u, v \in_R \mathbb{Z}_N$ and computes $U \leftarrow E(u)$, $V \leftarrow E(v)$, $X \leftarrow [u \boxdot B_j]$, $Y \leftarrow [v \boxdot A_j]$ and $Z \leftarrow [u \boxdot V]$. It sends (A_j, B_j, C_j) and (U, V, X, Y, Z) to all parties and gives a concurrent, non-malleable ZK proof of knowledge to each party of:
   - u such that $U \in E(u)$ and $X \in [u \boxdot B_j]$,
   - v such that $V \in E(v)$ and $Y \in [v \boxdot A_j]$, and
   - u such that $U \in E(u)$ and $Z \in [u \boxdot V]$.
2. Any $P \in \mathcal{P}$ receiving (A, B, C) and (U, V, X, Y, Z), along with accepting proofs, computes $A_{j+1} = A_j \boxplus U$, $B_{j+1} = B_j \boxplus V$, $C_{j+1} = C_j \boxplus X \boxplus Y \boxplus Z$, and sends a signature share on $\big((A_j, B_j, C_j), (P_k, gid, j, P_i), (A_{j+1}, B_{j+1}, C_{j+1})\big)$ to P_i.
3. P_i waits for $t+1$ valid signature shares on $\big((A_j, B_j, C_j), (P_k, gid, j, P_i), (A_{j+1}, B_{j+1}, C_{j+1})\big)$, computes a signature σ, and outputs $[(A_j, B_j, C_j), (P_k, gid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1})]$.

The following protocol allows the king (with help of the other parties) to generate a random multiplication triple (with gid *gid*). The idea is to start with an initial triple (i.e., encryption of $(1, 1, 1)$) and randomize it $t+1$ times—each time by a different party. For this the king first sends a randomization request for the initial triple to *every* party. Then he waits for the first correct answer and sends it as the second randomization request to all other parties (except the provider of the first randomization). Then again the first correct answer is used for the next randomization, etc. In every round, all but the first correct answers are ignored.

Protocol GenerateTriple.

0. P_k: Initialize $j = 0$ and $(A_0, B_0, C_0) = \big(E(1;\epsilon),\ E(1;\epsilon),\ E(1;\epsilon)\big)$.
1. For $j = 0$ to t do
   - 1.1 Send a randomization request $[P_k,\ gid,\ j,\ (A_j, B_j, C_j)]$ to every party P_i of whom no randomization for *gid* has been stored so far.
   - 1.2 P_i: Upon receiving a randomization request $[P_k, gid, j, (A_j, B_j, C_j)]$, employ the protocol RandomizeTriple to obtain $[(A_j, B_j, C_j),\ (P_k, gid, j, P_i, \sigma),\ (A_{j+1},\ B_{j+1},\ C_{j+1})]$, and send it to P_k. This is performed only once per *gid* and j.
   - 1.3 P_k: Upon receiving (from some party P_i for which no randomization for *gid* is stored so far) the first (correct) randomization answer $[(A_j, B_j, C_j), (P_k, gid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1})]$, store this answer. Further answers from other parties (for the same j) are ignored.
2. P_k: Send $[(A_j, B_j, C_j), (P_k, gid, j, P_{i_j}, \sigma_j), (A_{j+1}, B_{j+1}, C_{j+1})]$ for $j = 0, \ldots, t$ to every $P_i \in \mathcal{P}$, who accepts $(A, B, C) = (A_{t+1}, B_{t+1}, C_{t+1})$ as the final multiplication triple for *gid* if the following holds: For $j = 0, \ldots, t$, the j-th output triple is equal to $(j+1)$-th input triple, there are $t+1$ *different* parties that have randomized, and all transitions are correctly signed.

Given the multiplication triples (A, B, C) from GenerateTriple, and given encryptions X and Y to be multiplied, the following protocol computes an encryption of the product Z.

[4] Note that the correct value of each wire is well-defined, once the inputs are fixed.

Protocol Multiply.

0. Every P_i has input (A, B, C), X and Y.
1. P_i: send to P_k *and all slaves* decryption shares of $F = X \boxplus A$ and $G = Y \boxplus B$, and give proofs that the decryption shares are correct.
2. P_i and P_k: If $t + 1$ valid decryption shares for F and G arrive, compute $f = x + a \bmod N$ and $g = y + b \bmod N$ and let $Z = E(fg) \boxplus (-f \boxdot B) \boxplus (-g \boxdot A) \boxplus C$.

We first analyze the generation of the multiplication triple, then the multiplication protocol.

4.2 Analysis of Computation Phase

4.2.1 Analysis of GenerateTriple:

Although there is no agreement among the parties on the multiplication triple (A, B, C) (as such an agreement cannot be achieved with $t \geq n/3$) we are given certain guarantees about the triple (except with negligible probability):

- When an honest slave P_i accepts a triple (A, B, C), then A and B are encryptions of values a and b, and C is an encryption of ab. Furthermore, the set of corrupted parties (the adversary) cannot distinguish a and b from uniformly random values. This is formalized by a game, where the adversary has to distinguish $(D(A), D(B), D(C))$ from (a, b, ab) with uniformly random $a, b \in \mathbb{Z}_N$ with non-negligible advantage. Correctness follows from the correctness of the initial triple and the proofs of correct randomization. The indistinguishability follows from the fact that A and B result from $t + 1$ randomizations, so A and B were randomized by at least one honest party P_i. Therefore a and b sum over the u_i respectively the v_i contributed by P_i. Furthermore, every randomizing party P_j proves knowledge of its randomizers u_j and v_j (using a concurrent, non-malleable proof of knowledge, see Section 2.4). Hence we can, by rewinding, extract the u_j and v_j from the view of the adversary. So, for the adversary, distinguishing a and b from uniformly random is equivalent to distinguishing u_i and v_i from uniformly random for at least one honest P_i, which is impossible by the semantic security of the cryptosystem and the proofs given by P_i being concurrent zero-knowledge.

- When an honest slave P_i accepts a triple (A, B, C) for a gate *gid*, then the plaintexts of A and B are indistinguishable from uniformly random values which are *statistically independent* from the plaintexts of any triple accepted for any other gate $gid' \neq gid$. This does not follow from the above property which addresses the distribution of individual triples, but follows trivially from the fact that honest parties use different randomizers when contributing to different multiplication gates *gid*.

- When for the same multiplication gate *gid*, two honest parties accept the triples (A, B, C) and (A', B', C'), respectively, then either the plaintexts of (A, B, C) and (A', B', C') are indistinguishable from uniformly random, statistically independent values to the adversary, or the adversary knows the plaintexts of $A \boxminus A'$ and $B \boxminus B'$. This follows from the fact that either there is at least one honest party P_i that has randomized one triple in some position, but not the other one in the same position with the same (u_i, v_i) (then the plaintexts of the two triples are indistinguishable from uniformly random statistically independent values), or both triples have been randomized by exactly the same set of honest parties P_i in exactly the same positions with exactly the same (u_i, v_i). In this case only the adversarially chosen randomizers are different, and they are known to the adversary in the sense that they can be extracted from the adversary in expected polynomial time.

We now argue termination. Note that as long as at most t parties have randomized the triple, there are still $(n-t)-t \geq 1$ honest parties P_i which did not yet do so and thus, when requested, will eventually produce a randomization for *gid* and send it to P_k. Therefore, eventually a chain of $t + 1$ randomizations will be achieved.

4.2.2 Analysis of Multiply:

If P_k is honest, then all slaves will constantly agree on all ciphertexts X for each wire, and therefore the computation will terminate and will yield correct encryptions for all wires. When P_k is corrupted we do not *per se* care about the correctness of P_k, so what remains is to argue privacy.

The first important observation is that if an honest slave P_i associates X to some wire, then X is an encryption of the correct value for that wire. This holds for input wires by assumption and is maintained by addition. As for multiplication gate $z = xy$, we can assume that X and Y decrypt to correct values. If (A, B, C) was accepted by P_i as a correct triple, it is indeed a correct multiplication triple, except with negligible probability. From this it follows that if P_i computes some Z, then P_i computes a correct Z, except with negligible probability. Different parties might, however, hold different Z if P_k is corrupted—only the plaintexts are guaranteed to be the same.

We address the privacy. Assume that party P_i holds encryptions $X^{(i)}$ and $Y^{(i)}$ of the factors, and gets the multiplication triple $(A^{(i)}, B^{(i)}, C^{(i)})$ from P_k. At the same time, P_j holds encryptions $X^{(j)}$ and $Y^{(j)}$ of the factors, and gets the multiplication triple $(A^{(j)}, B^{(j)}, C^{(j)})$ from P_k. Then, P_k might learn the decryptions $f^{(i)} = x^{(i)} + a^{(i)} \bmod N$ and $g^{(i)} = y^{(i)} + b^{(i)} \bmod N$ as well as the decryptions $f^{(j)} = x^{(j)} + a^{(j)} \bmod N$ and $g^{(j)} = y^{(j)} + b^{(j)} \bmod N$. However, by the invariant that the values $X^{(i)}$ and $Y^{(i)}$ held by P_i (and the values $X^{(j)}$ and $Y^{(j)}$ held by P_j) encrypt correct wire values x and y, we have $x^{(i)} = x^{(j)} = x$ and $y^{(i)} = y^{(j)} = y$. Furthermore, from $(A^{(i)}, B^{(i)}, C^{(i)})$ and $(A^{(j)}, B^{(j)}, C^{(j)})$ being correct multiplication triples for the same gate *gid*, it follows that either 1) they encrypt values $(a^{(i)}, b^{(i)})$ and $(a^{(j)}, b^{(j)})$ which are uniformly random and independent, or 2) they encrypt values $(a^{(i)}, b^{(i)})$ and $(a^{(j)}, b^{(j)})$ which are individually uniformly random and $(a^{(j)}, b^{(j)}) = (a^{(i)}, b^{(j)}) + (\delta_a, \delta_b)$ for (δ_a, δ_b) *known*[5] to the adversary. In the first case, $(f^{(i)}, g^{(i)})$ and $(f^{(j)}, g^{(j)})$ are uniformly random and independent and thus together leak no information to the adversary. In the second case, $(f^{(i)}, g^{(i)})$ is uniformly random, and therefore leaks no information to the adversary,

[5]In the sense that we can extract them from the adversary in expected poly-time.

and $(f^{(j)}, g^{(j)}) = (f^{(i)}, g^{(i)}) + (\delta_a, \delta_b)$ and therefore leaks no more information than $(f^{(i)}, g^{(i)})$ to the adversary, as the adversary can compute it from $(f^{(i)}, g^{(i)})$ in expected poly-time.

4.3 Termination Phase

As for now no party can terminate until it knows that all honest parties for which it acts as slave terminated. However, this condition cannot be checked. Instead, we add the following simple procedure inspired by [CKS00] to terminate the protocol: When a king P_k learns the result z, it sends a signature share on ("`result`", z) to all parties and continues to act as slave. When it received signature shares from $t+1$ parties on ("`result`", z), it constructs a signature σ on ("`result`", z), sends (("`result`", z), σ) to all parties and terminates with output z. Any party ever receiving a value of the form (("`result`", z), σ) where σ is a valid signature on ("`result`", z) sends it to all parties, and terminates with output z. Eventually all $n - t \geq t + 1$ honest P_k learn z and thus some honest party eventually receives $t+1$ correct signature shares. After this all honest parties will eventually terminate.

5. ASYNCHRONOUS INPUT-DISTRIBUTION WITH $t < n/2$

In this section, we show how the input-distribution oracle can be reduced to an oracle that allows each party to synchronously broadcast one single message. More precisely, we construct a protocol for securely distributing inputs in an asynchronous network and $t < n/2$ faults, with a single invocation to a broadcast functionality.

Note that in an asynchronous network with $t \geq n/3$ without some additional oracle, input-distribution (even of a subset of parties) is impossible, as consistent distribution of a single input implies Byzantine agreement.

In the following, we show how *all inputs* can be distributed with $t < n/2$ with a single synchronous broadcast round.

In the input phase, every party P_i computes $X = E(x; r)$ for each of its inputs x and broadcasts X along with a ZK proof of plaintext knowledge (*PoPK*).[6] This ensures *input correctness*, in the sense that if P_i is honest, then $D_{dk}(X) = x$, and *input privacy*, in the sense that as long as at most t parties are corrupted, the input x of an honest P_i remains unknown to the adversary. This follows from the threshold IND-CPA security of the encryption scheme and the PoPK being ZK. Finally, the ZK proof of *knowledge* of x ensures *input knowledge*, meaning that P_i knows $D_{dk}(X)$ for his X. This is needed for the simulation.

The proof of plaintext knowledge could be based on standard assumptions by resorting to generic non-interactive zero-knowledge. In the following, we give a much more efficient proof, which exploits the fact that the proofs do not need to be fully non-interactive, but asynchronous interaction (with $t < n/2$) is allowed for verifying the proof. We call such proofs *almost non-interactive proofs*.

The intuition of our almost non-interactive proof is the following: The prover sends along with the encrypted input a *transcript* of many instances of an interactive zero-knowledge proof of plaintext knowledge with binary challenges. For each instance, the prover provides the answers for *both challenges*, but encrypts them with the threshold encryption scheme. To verify the proof, for each instance exactly one response (depending on an agreed-upon challenge) is decrypted. The challenge is generated simultaneously, by letting every party P_i broadcast an encryption R_i of a random value r_i, where the encryption scheme has the property that both the parties jointly as well as P_i alone can decrypt.[7] Then the parties decrypt all contributions and compute the challenge r as the sum.

In the next section, we describe how to generate the random challenge (using almost non-interactive verifiable secret-sharing). Subsequently, we describe in more detail how to construct the almost non-interactive zero-knowledge proof of plaintext knowledge.

5.1 Almost Non-Interactive Verifiable Secret-Sharing

The following protocol allows a sender P_S to verifiably secret share a secret x with threshold $t < n/2$, using a single round of synchronous broadcast. The reconstruction of the shared value is fully asynchronous. We call this *almost non-interactive verifiable secret-sharing* (*ANI-VSS*).

The ANI-VSS requires a setup—for every P_S there is an independent random key pair (pk_S, sk_S) for a threshold cryptosystem such that the public key pk_S is known to all parties and the secret key sk_S is shared among the other parties with threshold t (such that correct decryption shares from $t+1$ parties are enough to decrypt under sk_S). We also require that P_S knows sk_S. If not already the case, this can be ensured by all parties once-and-for-all sending their shares of sk_S to P_S.[8] The protocol proceeds as follows:

Synchronous sharing: P_S computes $X \leftarrow E_{pk_S}(x)$ and broadcasts X (using synchronous broadcast).

Asynchronous reconstruction:

1. Each P_i computes a decryption share of X using his share of sk_S and sends the share to all parties along with a proof of correctness.
2. Each P_j waits for $t+1$ correct decryption shares and reconstructs $x = D_{sk_S}(X)$.

5.1.1 Analysis:

We assume that the encryption schemes have perfect decryption. This means that the broadcasted message X uniquely defines a secret $x = D_{sk_S}(X)$. Reconstruction will always terminate as at least the $n - t \geq t + 1$ honest parties send correct decryption shares.

In terms of simulation security, an ANI-VSS is extracted by decrypting X. This is possible as the honest parties hold enough decryption key shares to compute sk_S. When P_S is honest, an ANI-VSS is opened to any x' simply by simulating the decryption of X to hit x'.

5.2 Almost Non-Interactive ZKPoK

We now describe a system which allows a prover P to give a ZK proof of knowledge (*ZKPoK*) towards all parties such that all parties agree on the outcome of the proof. The protocol uses only one round of synchronous broadcast,

[6]The details of the PoPK are given below.

[7]This way no PoPK for R_i is necessary.

[8]In fact, the fact that they *could* do this is sufficient for the analysis.

followed by an asynchronous computation. We call it an *ANI-ZKPoK*.

We consider some NP relation R and assume that P holds an (instance, witness)-pair (x, w). We assume that there is a standard three-move Σ-protocol for R, where P computes the first message a, gets a challenge $e \in \{0,1\}$ and replies with some response z. The verifier accepts or rejects based on (x, a, e, z). We use that from two accepting conversations $(x, a, 0, z_0)$ and $(x, a, 1, z_1)$ one can compute (in PPT) a witness w such that $(x, w) \in R$. The protocol proceeds as follows:

Synchronous proof: The synchronous round proceeds as follows:

- Prover P: For $k = 1, \ldots, \kappa$, compute a first message $a^{(k)}$ and a reply $z_0^{(k)}$ to the challenge $e = 0$ and a reply $z_1^{(k)}$ to the challenge $e = 1$. Then broadcast x and each $a^{(k)}$ and ANI-VSS each $z_0^{(k)}$ and $z_1^{(k)}$.
- Each other party P_i: ANI-VSS a uniformly random value $r_i \in \{0,1\}^\kappa$.

Asynchronous verification: The verification of the proof is asynchronous, and proceeds as follows:

1. Reconstruct each r_i and compute $(e_1, \ldots, e_\kappa) = \oplus_{i=1}^n r_i$.
2. For $k = 1, \ldots, \kappa$ in parallel: Reconstruct $z_{e_k}^{(k)}$ and accept the proof if and only if $(a^{(k)}, e_k, z_{e_l}^{(k)})$ is an accepting conversation for $k = 1, \ldots, \kappa$.

5.2.1 Analysis:

After the first (synchronous) part, all parties will hold consistent proof transcripts $(a^{(1)}, Z_0^{(1)}, Z_1^{(1)}), \ldots, (a^{(\kappa)}, Z_0^{(\kappa)}, Z_1^{(\kappa)})$ (as broadcasted by the prover) as well as consistent encryptions of challenge-contributions R_i of every party P_i (as broadcasted by P_i). It follows that in the asynchronous part all parties will reconstruct the same $r_1, \ldots r_n$ leading to the same $(e_1, \ldots, e_\kappa)$ and thus leading to the same outcome of the verification test. It is clear that if the prover is honest, this outcome will be accepting. Since each reconstruction eventually terminates, the proof eventually terminates.

Assume that P broadcasted $(a^{(1)}, Z_0^{(1)}, Z_1^{(1)}), \ldots, (a^{(\kappa)}, Z_0^{(\kappa)}, Z_1^{(\kappa)})$ without knowing a witness for x. Then for each k there exists e'_k such that $(a^{(k)}, e'_k, z_{e'_k}^{(k)})$ is not accepting.[9] Thus there is at most one challenge $e = (e_1, \ldots, e_\kappa)$ for which the verification of the proof is not rejecting, namely $(1 - e'_1, \ldots, 1 - e'_\kappa)$. As the prover had to choose and broadcast $(a^{(1)}, Z_0^{(1)}, Z_1^{(1)}), \ldots, (a^{(\kappa)}, Z_0^{(\kappa)}, Z_1^{(\kappa)})$ without knowing the r_i's of the honest parties (and thus without knowing the resulting challenge e), his success probability is negligible.

[9]Contra-positively, if both encrypted conversations are valid, then P can use his knowledge of the secret key to learn the two valid conversations $(a^{(k)}, 0, z_0^{(k)})$ and $(a^{(k)}, 1, z_1^{(k)})$ in poly-time and can then compute from these a valid witness w in poly-time, which by definition means that he knows w.

To simulate a proof, the simulator will for each k use the honest verifier simulator of the Σ-protocol to prepare a uniformly random bit e_k for which it knows a valid conversation $(a^{(k)}, e_k, z_{e_k}^{(k)})$. It lets $z_{1-e_k}^{(k)} = z_{e_k}^{(k)}$. I.e., it can answer only e_k correctly. Then it lets $e = e_1 \ldots, e_\kappa$. Then the simulator extracts the r_i contributed by corrupted parties from the ANI-VSS's. Finally the simulator forces the coin flip to hit exactly the challenge e which it can answer by opening the ANI-VSS for an honest P_j to $r_j = e \oplus \bigoplus_{i \neq j} r_i$.[10]

6. CONCLUSIONS AND OPEN PROBLEMS

We presented an asynchronous protocol which evaluates any agreed function on predistributed inputs, securely against an active adversary corrupting $t < n/2$ parties. This is the first asynchronous MPC protocol for $t \geq n/3$. We stress that all previous asynchronous MPC protocols (and in particular the multiplication sub-protocols) inherently require $t < n/3$, even once inputs are distributed.

Furthermore, we have presented an asynchronous protocol for distributing the inputs with $t < n/2$, assuming an oracle which allows every player to synchronously broadcast one single message. We stress that asynchronous input distribution with $t \geq n/3$ is not possible without the help of an oracle. Furthermore, our protocol for input distribution takes further advantage from the broadcast oracle and guarantees that all parties can provide input. This is provably not possible when no such oracle is available.

These results can be brought together to an asynchronous MPC protocol with $t < n/2$, for a model where the first communication round is synchronous. Alternatively, it can be interpreted as an asynchronous MPC protocol which tolerates $t < n/3$ faults in the first rounds, and then $t < n/2$. Also, the result can be compared with the synchronous world (without setup), where MPC is possible for $t < n/3$ when no broadcast is available, and $t < n/2$ when black-box broadcast is available.

This work leaves a lot of interesting open problems: First of all, our protocol requires quite strong setup assumptions, and it is not clear whether they are necessary. Furthermore, the provided protocol provides cryptographic security only. When e.g. perfect security is required, then MPC requires $t < n/3$ in the synchronous case, respectively $t < n/4$ in the asynchronous case. We do not know whether this gap also stems solely from the input distribution, i.e., whether it is possible to evaluate any function of predistributed values over an asynchronous network perfectly secure with $t < n/3$.

[10]For the ANI-VSS described above, the extraction would simply amount to decryption under the secret key used by P_i. The honest parties have enough shares to facilitate this.

7. REFERENCES

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 52–61, 16–18 May 1993.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.

[BH07] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous mpc. In *Advances in Cryptology - ASIACRYPT 2007*, Berlin, 2007. Springer. Lecture Notes in Computer Science.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computation with optimal resilience. In *Proc. ACM PODC '94*, pages 183–192, 1994.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 87–119, Berlin, 1987. Springer-Verlag. Lecture Notes in Computer Science Volume 293.

[CDN01] Ronald Cramer, Ivan Damgaard, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology — EUROCRYPT 2001*, pages 280–300, Berlin, 2001. Springer-Verlag. Lecture Notes in Computer Science Volume 2045.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 123–132. ACM, July 2000.

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In *EUROCRYPT*, pages 418–430, 2000.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In D. Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, pages 247–264, Berlin, 2003. Springer-Verlag. Lecture Notes in Computer Science Volume 2729.

[FH96] Matthew Franklin and Stuart Haber. Joint encryption and message-efficient secure computation. *Journal of Cryptology*, 9(4):217–232, Autumn 1996.

[FN09] Matthias Fitzi and Jesper Buus Nielsen. On the number of synchronous rounds sufficient for authenticated Byzantine agreement. In Idit Keidar, editor, *Distributed Computing, 23rd International Symposium, DISC 2009*, volume 5805 of *Lecture Notes in Computer Science*, pages 449–463. Springer-Verlag, 2009.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, 25–27 May 1987.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience (extended abstract). In R.Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, pages 322–340, Berlin, 2005. Springer-Verlag. Lecture Notes in Computer Science Volume 3494.

[Nie02] Jesper Buus Nielsen. A threshold pseudorandom function construction and its applications. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 401–416, Berlin, 2002. Springer-Verlag. Lecture Notes in Computer Science Volume 2442.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residue classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, 1999. Springer-Verlag. Lecture Notes in Computer Science Volume 1592.

[Sho00] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220, Berlin, 2000. Springer-Verlag. Lecture Notes in Computer Science Volume 1807.

[Tou84] Sam Toueg. Randomized Byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 163–178, 1984.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, 3–5 November 1982. IEEE.