The SPDZ Protocol or:

Secure Computation from Somewhat Homomorphic Encryption and Unconditionally Secure MACs

Winter School, Bar Ilan, 2015

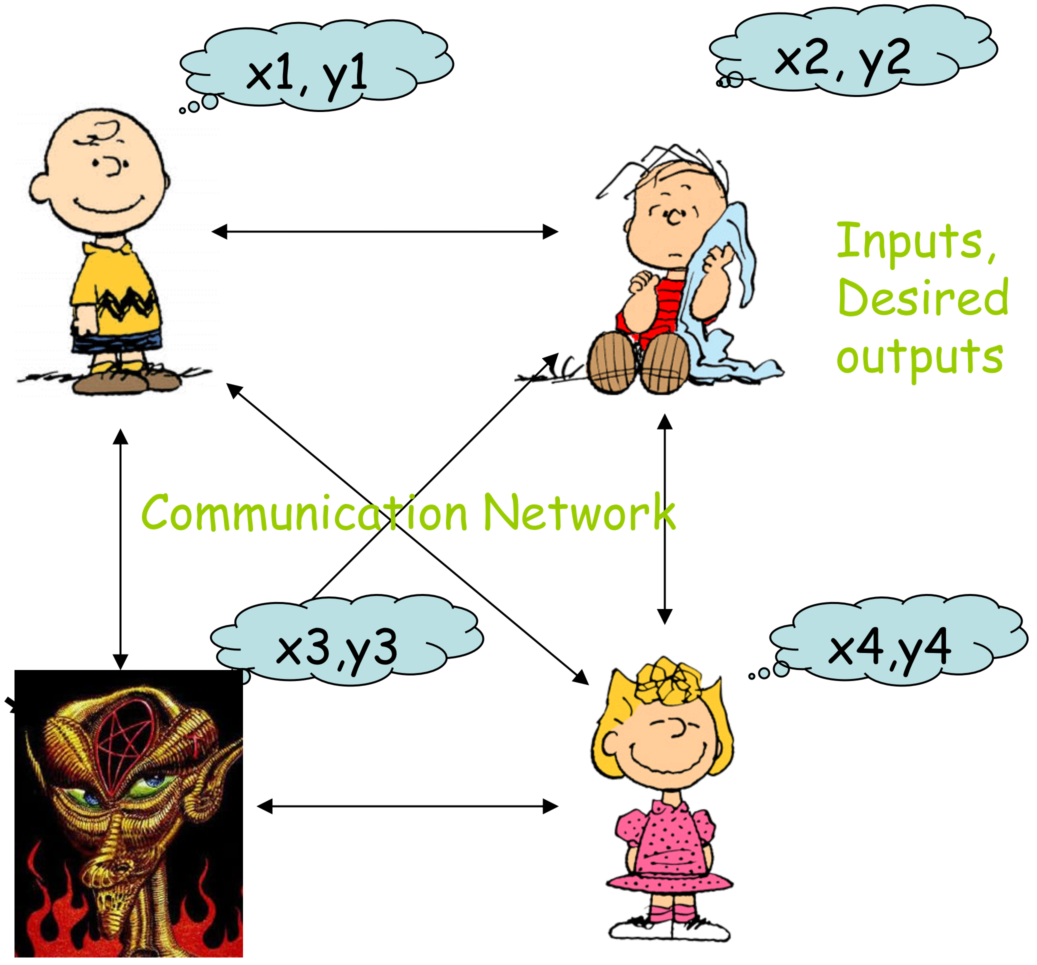
Ivan Damgård

Dept. Of Computer Science, Århus University

The MPC scenario

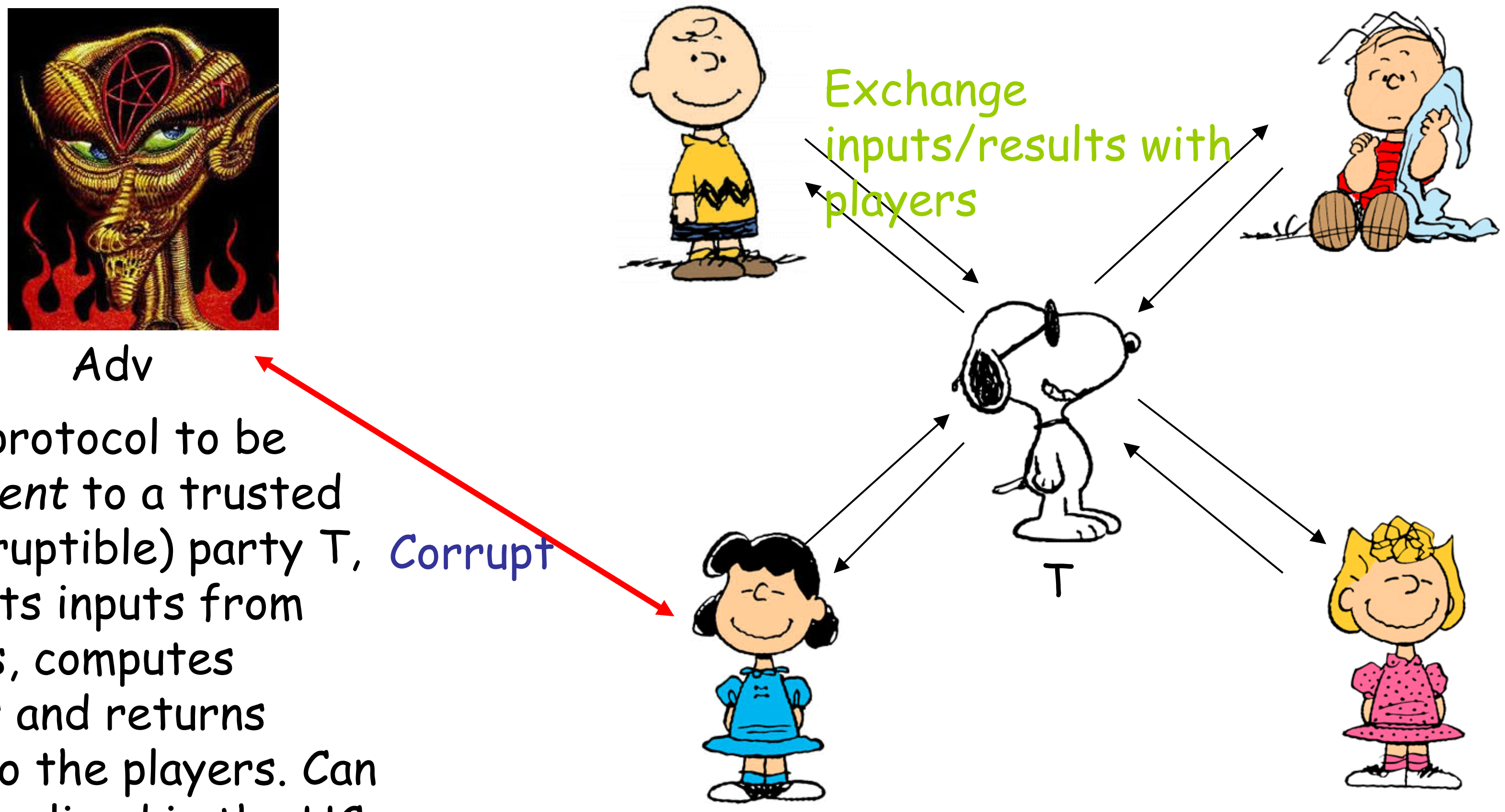
Corruption can be *passive*: just observe computation and mess.

In this talk: *active*: take full control



Also, we assume throughout static corruption: Adversary must corrupt initially.

Goal of MPC – a bit more precisely



Want protocol to be *equivalent* to a trusted (uncorruptible) party T, who gets inputs from players, computes results and returns them to the players. Can be formalized in the UC model.

Known Results on Information Theoretic MPC

General flavor of known results: as long as not too much corruption happens, any function can be securely computed. If there is too much, some functions become impossible to handle (usually includes the most interesting ones).

In particular:

Assuming secure point-to-point channels (plus broadcast) and honest majority, can get unconditional security for any function – and quite efficiently too.

The Case of Dishonest Majority

Assume t=n-1 players can be corrupted. Cannot have unconditional security.

But we can protect against a computationally bounded adversary using public-key cryptography:

If t=n-1, we can compute any function securely, based on suitable computational assumptions (but termination cannot be guaranteed).

So the efficient solutions for unconditional security are useless here?

NO!

"MPC in the head" can improve efficiency [IKOS06, etc.]

"MPC with Preprocessing" this talk. Background:

[Bendlin Damgård Orlandi Zakarias EC11], (BeDoZa)
[Nielsen Nordholt Orlandi 11] (TinyOT), and
[Damgård Pastro Smart Zakarias 11] (SPDZ -1),
[Damgård Zakarias 12], (MiniMac)
[Damgård Keller Laraia, Pastro, Scholl, Smart 13] (SPDZ-2)

Reminder: MPC with Preprocessing

• Preprocessing Phase, can be done any time, inputs, function to compute need not be known. Create raw material for

• On-line phase, inputs, function supplied here. Function can be computed more efficiently because we have raw material from preprocessing.

Goal: use only cheap information theoretic tools here, push all the expensive public-key stuff into preprocessing.

In the following..
Think of the Preprocessing as Trusted Dealer, whom we can ask to give us data in any form we like.
Later, must implement the dealer via secure protocol that does not assume trusted parties..

Reminder: Semi-Honest Solution (Players are honest but curious)

Secret $x = x1+x2 \bmod p$, p a prime, x1,x2 shares in x

x1

x2

x1, x2 could be supplied by the dealer. Then neither player has information on x.
But players can open x by exchanging shares.

Will use field with p elements here, but everything works for finite fields in general..
Will consider 2 players for now, but everything works for n>2 players as well..

Computing on shared values.

[x] = (x1, x2)

[y] = (y1 y2)

Can define [x] +[y] = [x+y] by local componentwise addition.

Similar for mult. by public constants.

Now we can compute any linear function on shared values.

Multiplication [Beaver's Circuit Randomization Technique]

Given [x], [y], want to compute [xy]

Assume the trusted dealer will give to the players

[a], [b], [c], random a,b, c=ab.

Open e= x-a,   d= y-b

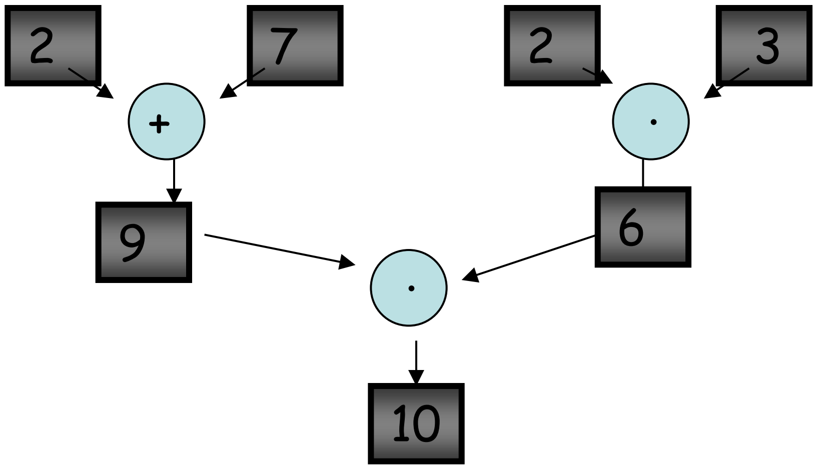
Then

[xy] = [c] + e[b] + d[a] + ed

A trusted dealer can also help us make representation of an input a known to only player A:

Assume dealer has supplied [r], for random r and has given r to A.
- A adjusts his share of [r] by adding a-r to it.
- Means we now have [a].

Putting it all together



Circuit and inputs given

Create "objects" of form [a] representing inputs. Jointly held by players, value not accessible to adversary.

Compute new objects.

Open outputs

Conclusion

Can do general passively secure multiparty computation, assuming that preprocessing issues enough triples [a], [b], [c] with c= ab.

What if Players do not follow the protocol?

Previous solution does not work: players can lie about their shares
-> we may compute the wrong function and data can leak.

The solution you have seen:
Rabin/Ben-Or[RB88], Bendlin et al. [BDOZ, EC11], also [Nielsen et al. "TinyOT" Crypto12]: Authenticate shares. Charlie's share comes with a message authentication code (MAC), Lucy has a corresponding key.
When Charlie reveals his share, must produce MAC, Lucy checks using her key. To lie successfully, Charlie must guess the key.

Problem: this does not scale well with number of players n: requires each player to store O(n) field elements.

The idea of SPDZ [Damgård, Pastro, Smart, Zakarias 11]

Authenticate the secret value itself, not the shares
- stay tuned for details, first

The authentication scheme we will use:
Message x, Key α, MAC is m(x) = α x (mult in the field we use)

Security game: Adversary sees x (but not the key nor the MAC), chooses error contributions: e to modify x and e' to modify MAC.

A verifier checks that m(x) = α (x+e) + e'
Adversary wins if check goes through and e is not 0.

If check goes through, then we have: m(x) = α x = α (x+e) + e'
Equivalently: αe + e' =0
If e is not 0, this determines α. But e, e' are chosen independently of α, so adversary wins with probability 1/p.

Nice homomorphic property: the sum of two MACs is a MAC on the sum of the messages.

A new representation of secret values

Secret value x= x1+x2 mod p, x1,x2 shares in x

MAC $m(x) = \alpha x = m(x)_1 + m(x)_2$ mod p shares in the MAC

In addition dealer will issue shares α1, α2 to players such that α =α1+ α2.

x1 x2

$m(x)_1$ $m(x)_2$

Now, one player may be malicious. So not clear how we open a value reliably. Of course the MAC plays a role, but we cannot reveal the key α, the adversary could then cheat all other MACS.

Will handle this later..

Computing with Representations.

[x] = (x1, $m(x)_1$ x2, $m(x)_2$)

[y] = (y1, $m(y)_1$ y2, $m(y)_2$)

Can define [x] +[y] = [x+y] by local componentwise addition.
Same for multiplication by public constants.

Add public constant d to [x]:
d+[x] =
[x+d]= (x1+d, $m(x)_1$ + d α1 x2, $m(x)_2$+ d α2)

Can be done locally since players have shares of α.

Computing with representations, continued.

$[r] = (r1, m(r)_1$ $\qquad\qquad$ $r2, m(r)_2)$

To enter input x held by Charlie, assume Dealer supplies [r] and r itself to Charlie. He broadcasts x-r, and players compute (x-r) + [r] = [x] (previous slide showed how to add public constant.

Multiplication done using random multiplication triplets [a], [b], [c] supplied by dealer, just as in passive case.

Protocol for Opening values.

Recall: Secret value x= x1+x2 mod p, x1,x2 shares in x

MAC m(x) = α x = $m(x)_1 + m(x)_2$ mod p shares in the MAC

Dealer issues shares α1, α2 to players such that α =α1+ α2.

1) Players exchange shares of x and add. A corrupt player may lie and add an error e his share, so we get x´= x +e.
2) Players compute locally
   d1= α1 x'- $m(x)_1$ and d2= α2 x' - $m(x)_2$
   Note that d1 +d2 = α x'– m(x)
3) Players each commit to d1 and d2, and then open (Assume for now an ideal commitment functionality). Players add the opened values and accept if sum is 0.

A dishonest player may commit to a wrong value, so we check that
0= d1 + d2 + e' = α x'– m(x) +e' = α (x+e) – m(x) +e'
Or equivalently, m(x) = α (x+e)+e'.
Equivalent to security game for MACs: if e is not 0, this happens with probability 1/p.

But how to do commitments for real?

Extended representation of secret values

Secret value $x= x1+x2 \bmod p$, $x1,x2$ shares in x

$m1(x) = \beta1\, x = m1(x)_1 + m1(x)_2 \bmod p$

$m2(x) = \beta2\, x = m2(x)_1 + m2(x)_2 \bmod p$

Dealer issues keys $\beta1$, $\beta2$ to players.

$x1$

$m1(x)_1, m2(x)_1$

$\beta1$

$x2$

$m1(x)_2, m2(x)_2$

$\beta2$

Notation: [[x]] for extended representation.

Clearly, can open reliably: just send your share of x and of the MAC that the other player(s) can check. Works by same proof as before.
Requires more data than [x], so use with care!

Protocol for Commitments.

Assume: Dealer supplies [[r]] to all players and r to the committer P.

1) P wants to commit to value x. P broadcasts x-r.
2) To open, all players open [[r]] and all can compute x= (x-r)+r

Clearly, this is information theoretically secure.

But for n players, will require to store $O(n^2)$ data values per commitment. Much more expensive than the [] – representation. So we should minimize the use of commitments.

On-line Protocol, first version.

1) Establish representation [x] for each input x.

2) Go through Arithmetic circuit gate by gate and do addition, add constant, multiply by constant or multiply protocol, according to the type of gate encountered.

3) We end with representations of outputs, which we open.

This is secure, but efficiency is bad: each multiplication requires that we open 2 values, requires 2 commitments by each player each time. Way too expensive.

Online Protocol, Second version.

Instead of checking MACs individually for all opened values, do *partial opening* where only shares of data values are exchanged. No MAC check just now.

1) Establish representation [x] for each input x.

2) Go through Arithmetic circuit gate by gate and do addition, add constant, multiply by constant or multiply protocol. In multiply protocol, do only partial openings.

3) We end with representations of outputs.

4) Do a batch-check of macs on all partial openings in 2), as described on next slide.

5) If check in 4) was OK, open outputs using normal opening protocol.

Cheap Batch-checking of many MACs

We have [x0], [x1],…,[xt] and opened values x'0,…, x't
Want to check whether xi= x'i

Instead of checking individually, take a random
linear combination – choose e0,…,et at random and compute (locally)

[y] = [e0 x0 + … + et xt] as well as y'= e0 x'0 +…+ et x't.

If any x'i is different from xi, y'=y with probability at most 1/p.

Now we check that y'=y, as in opening protocol from before:
Pi computes locally di= share of MAC(y) – y' times share of α,
commits to di and open later. Sum must be 0.

If y is not y', this check will accept with probability at most 1/p.

A final problem with MAC checks.

Choosing all the ei's at random is problematic: who does the choice? Easy solution: have preprocessing supply [[e_i]] for all i. Works, but uses way to many [[]]- representations.

Instead, just one random value [[u]] will suffice.
When the time comes, we open u and define ei= u^i mod p

Recall
[y] = [e0 x0 + … + et xt] as well as y'= e0 x'0 +…+ et x't.

Now, if y=y', we have

0= y-y' = (x0-x'0) u^0 + … + (xt-x't)u^t

If some xi-x'i is not 0, then check only works if u happens to be a root in a non-zero polynomial of degree at most t.

Error probability is now t/p.

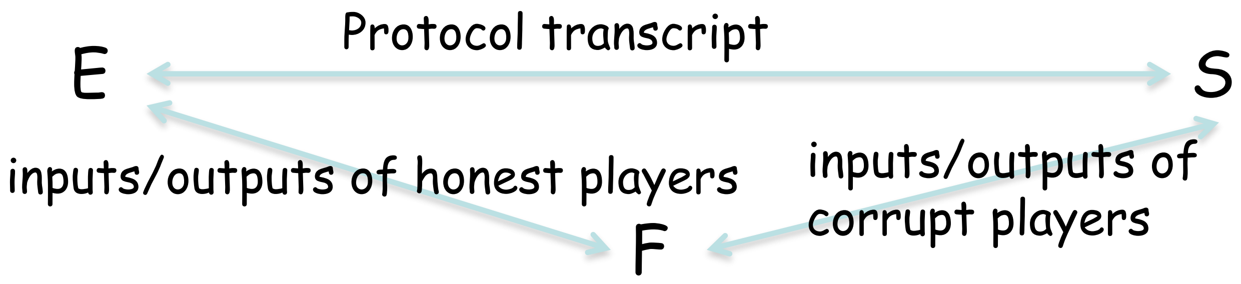
How to prove Security.

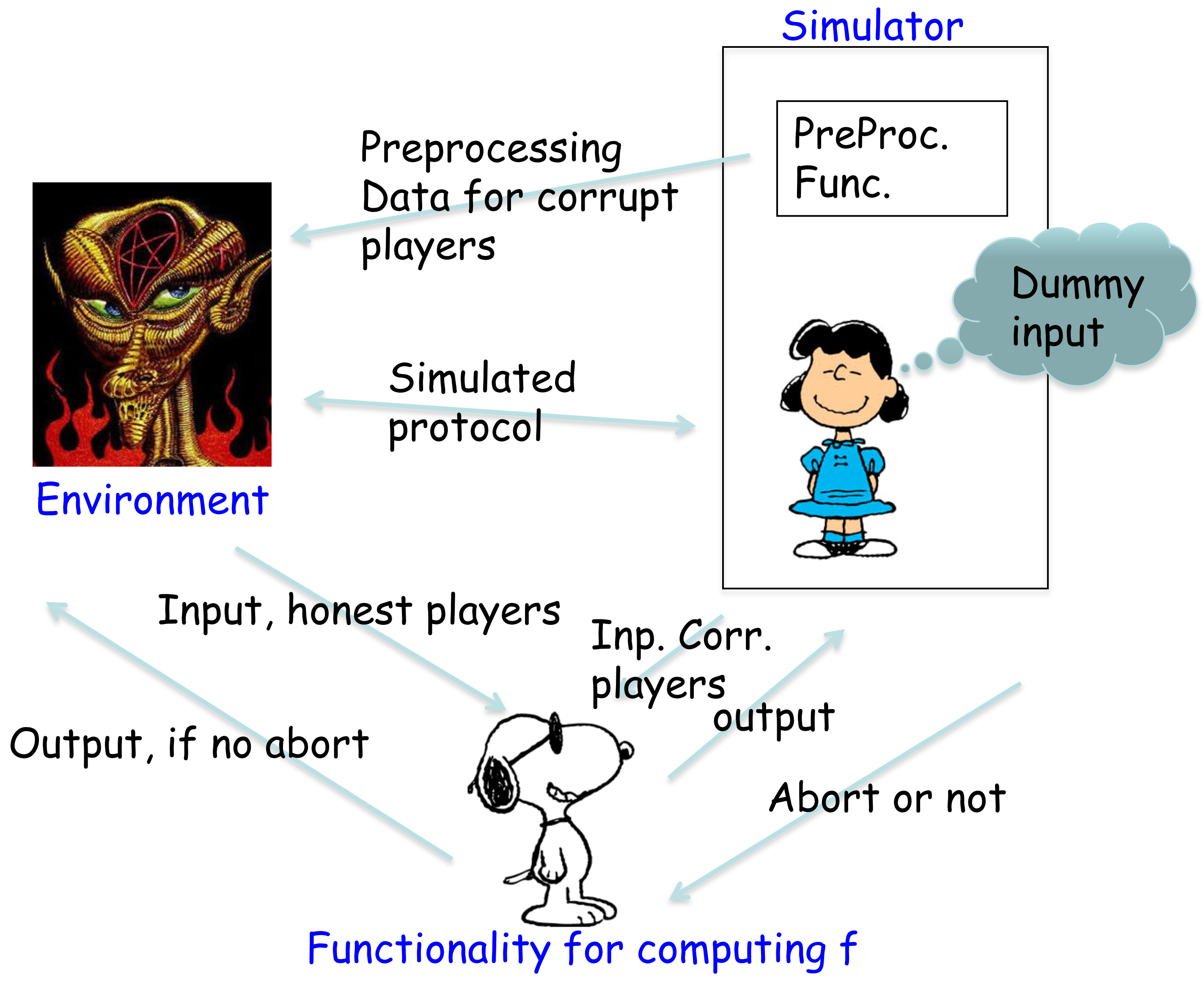
Need to construct a simulator S, talks to environment (who represents corrupted players) on one side and to Ideal Functionality F on the other side.

Here F is the functionality that

1) gets inputs from players,
2) computes desired function
3) sends output for corrupted players to adversary
4) If adversary says OK, send output to honest players

Step 3) needed because we have dishonest majority.


Protocol transcript
E
S
inputs/outputs of honest players
inputs/outputs of corrupt players
F

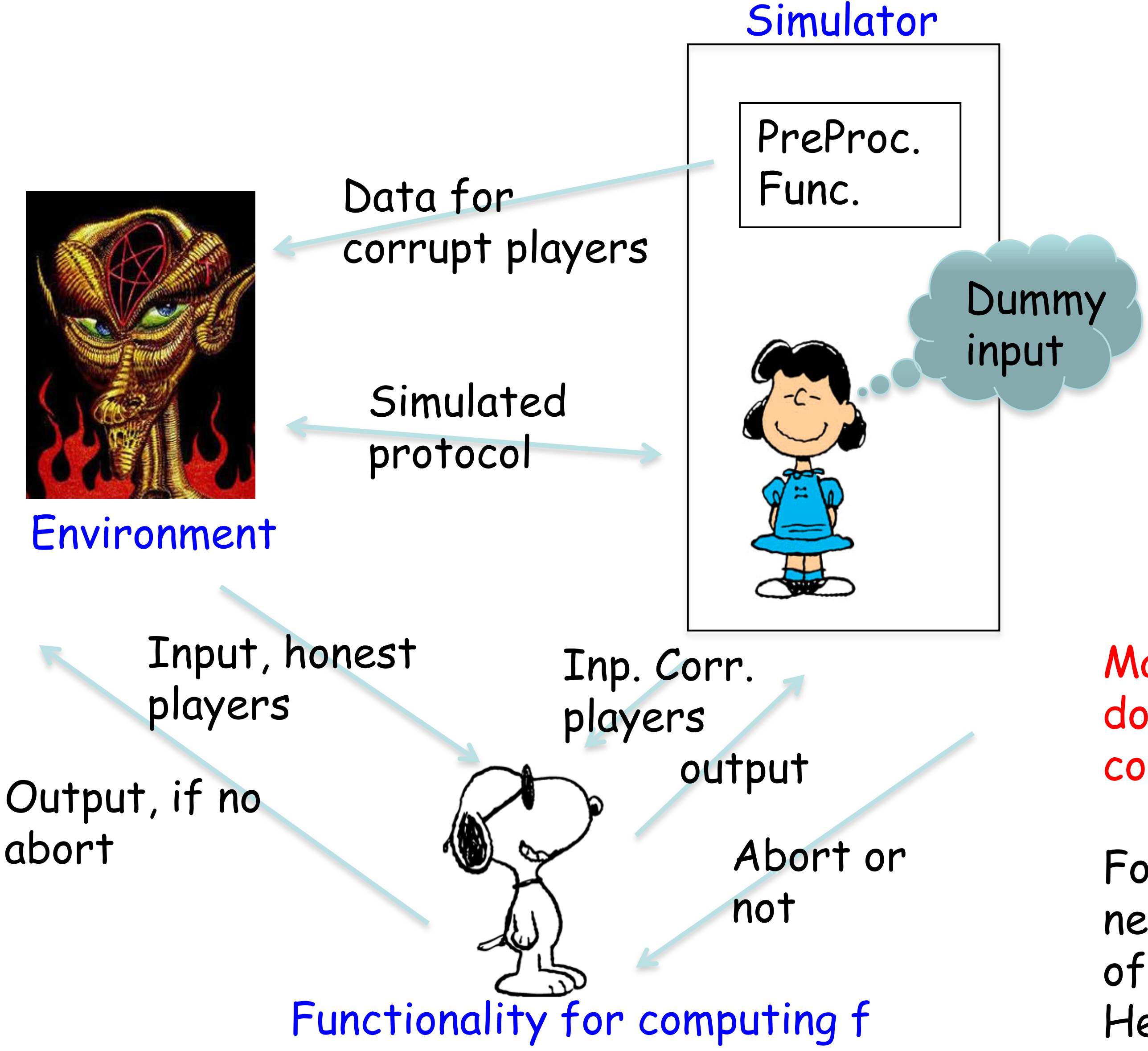

Simulator
PreProc. Func.
Dummy input
Preprocessing Data for corrupt players
Simulated protocol
Environment
Input, honest players
Inp. Corr. players
output
Output, if no abort
Abort or not
Functionality for computing f

Sketch of S.

Protocol assumes functionality for preprocessing, so S needs to also simulate what that functionality sends to corrupt players initially.
S starts its own copy of each honest player with default inputs, say 0.
Can simulate what honest players send to corrupt players by just following the protocol. Additional issues:

1)Input stage: When corrupt player P sends value c, this is supposed to be c= x-r for an r that S generated earlier. S sends c+r to F as P's input.
2)Computation phase: just follow protocol.
3)Output phase: S gets output(s) y from F. Currently have representation of different value [y']. S adjusts the share of an honest player so that we now have [y]. S can do this because it knows the global MAC key. Then we execute the normal opening.
4)If all checks are OK in the simulated protocol, tell F it's OK to release outputs.

$$\begin{tikzcd}
E \arrow[rr, leftrightarrow, "\text{Protocol transcript}"] \arrow[dr, leftrightarrow, "\text{inputs/outputs of honest players}"'] & & S \arrow[dl, leftrightarrow, "\text{inputs/outputs of corrupt players}"] \\
& F &
\end{tikzcd}$$



This generates a view for E with a correctly sampled protocol transcript where the correct output is computed.

What's not to like?

Maybe real protocol does not compute correct output!

Fortunately, almost never happens because of the MACs
Hence statistical sec.

Efficiency – making it scale well with number of players and circuit size.

The good news: for each representation [x] we have, each player stores only a constant number of field elements. And required number of representations is linear in circuit size. Number of expensive [[-]] –reps. independent of circuit size.

Not so good news: when we open, each of the n players send a share to all other players, so we have $O(n^2)$ messages.
Solution: all players send shares to a single player P, he adds them and broadcasts result to all. Now only O(n) messages and additions.
P might lie about result, but this is equivalent to a corrupt player lying about his share in the first place.

But isn't Broadcast expensive?

Yes, but we can make do with less: the overall protocol cannot guarantee termination, so the broadcast doesn't have to either..

“Discount” solution for broadcast.

Say P wants to broadcast value x.
1) P sends x to all players.
2) All players send what they received to all players
3) Each player compares all values he has seen. Abort if any mismatch.

If two honest players received different values in step 1), then at least one honest player will abort. All honest players will see this in the next round and will abort too.
We send $O(n^2)$ field elements, but can do better (amortized):

Say P wants to broadcast values x1,.., xn.
1) P sends x1,…, xn to all players.
2) All players send a hash of what they received to all players.
3) Each player compares all hashes he has seen. Abort if any mismatch.

We can use a universal hash function that maps into the field. Gives unconditional security and amortized we send O(n) field elements to broadcast one element.

Summing up On-line Phase

For n players, to compute arithmetic circuit of size S securely with error probability ≈ 1/p, each player needs to

- store $O(S + n^2)$ field elements.
- do $O(S + n^2)$ local field operations.

For large circuits and large fields, where 1/p acceptable error probability, means each player spend not much more work than you need to compute the circuit with no security!

Implementation: 3 players, connected on a LAN,
64 bit prime p, one secure multiplication in
amortized time about 0.003 ms.

Optimality of On-line Phase.

[Winkler and Wullschleger, Crypto 10] For 2-party protocols in the preprocessing model, show lower bounds for the amount of preprocessed data you need to compute a function f securely with stat. security.
Bound depends on certain combinatorial properties of f, works for OT, for instance. Suppose Pi gets input bit y and all other players have strings x and x' of length S field elements.
Pi is supposed to learn $yx + (1-y)x'$. A circuit of size $O(S)$ is sufficient.

A multiparty protocol in preprocessing model where each player needs to store $o(S)$ field elements, would imply a 2-party protocol contradicting WW.
So a protocol that allows computing any circuit of size S must have players store $\Omega(S)$ field elements.

Can also show that Pi will have to read all his preprocessed data.
Implies lower bound of S field operations for the work of Pi.
Open problems: can we show that ALL players have to work this hard simultaneously? What about communication complexity?

Implementing the Dealer, or: Preprocessing

Assume a somewhat homomorphic encryption scheme allowing many values to be multiplied in parallel inside one ciphertext.

Cryptosystem based on the Ring-LWE assumption, variant of [Brakersky et al.], Crypto 2011, using Smart et al.'s SIMD extension.

Somewhat homomorphic Cryptosystem: A secure public-key scheme, plaintexts are vectors $\mathbf{m}$= (m1,…,ms) in Z_p^s

$E_{pk}(\mathbf{a}+\mathbf{b}) = E_{pk}(\mathbf{a}) + E_{pk}(\mathbf{b})$

$E_{pk}(\mathbf{a}*\mathbf{b}) = E_{pk}(\mathbf{a})\, E_{pk}(\mathbf{b})$ where $\mathbf{a}*\mathbf{b}$ is coordinatewise multiplication

- and multiplication of ciphertexts takes place in some ring.

$D_{sk}(E_{pk}(\mathbf{x})) = \mathbf{x}$

- if $E_{pk}(\mathbf{x})$ is not "too dirty", i.e. the result of "too many" operations.

In our case: "not too many"= 1 multiplication and many additions.

Just what we need for preprocessing: each multiplication triple can be prepared using the same set of operations, can all be done in parallel.

Distributed Decryption.

Can think of ciphertext c as two polynomials (c0, c1) in
Zq[X]/(f(X))
Where q >> p and f is some cleverly chosen poynomial.

Simplistically speaking, secret key is a single polynomial s.
For ciphertext (c0,c1),
(c0- s c1) mod p
is the plaintext (polynomial), one reads off the coefficients
to get message.

We can split s as s= s1 +… + sn and give si to player Pi

To decrypt, Pi sends di= -si c1 + ti
where ti mod p=0 and ti has relatively small coefficients.
ti is "extra noice" that is there to "hide" si.

Now, the message is
(c0- (d1+…+dn)) mod p

This is semi-honest secure.

Distributed Decryption

Previous protocol allows a malicious adversary to modify result.

But it is a secure implementation of a functionality that

- Takes ciphertext as input, decrypts,
- Sends message to adversary
- Allows adversary to decide output.

This will be sufficient, if we design the rest of the protocol such that we can catch the errors introduced.

Set-up Assumption (some set-up necessary for UC security)
A public key has been generated, and shares of the secret key given to players.

(Sketch of) Preprocessing Protocol

Assume for simplicity that ciphertexts contain just a single field value.

1)Each player Pi chooses ai, bi, ri at random and broadcasts
Ai= E_{pk}(ai), Bi= E_{pk}(bi), Ri= E_{pk}(ri). Gives a ZK proof that he knows the plaintexts.
2)All players compute A= A1+…+An, B= B1+…+Bn, and D= AB. Define a to be sum of the ai's, likewise with b. D then contains c= ab, but is a "noisy" ciphertext (result of a multiplication).
3)Use distributed decryption to decrypt D-R, get result d (= ab-r+e).
4)Set C= R+ E_{pk}(d) If all went well, then C contains ab. In general it contains ab +e where e is an error known to the adversary.
5)P1 outputs a1, b1, c1= r1+d, other Pi output ai, bi, ci= ri

Note that the ci sum to c+e. We deal with this error later..

This looks already like representations [a], [b], [c]. But we still need the MACs..

How to add MACs

0) Once and for all: create encryption $K = E_{pk}(\alpha)$ for random α, as on previous slide.

Given a ciphertext $X = E_{pk}(x)$, want to compute additive shares in the MAC on x.

1)Each player Pi chooses si at random and broadcasts $Si= E_{pk}(si)$. Gives a ZK proof that he knows the plaintext.
2)All players compute S= S1+…+Sn, and KX.
3)Use distributed decryption to decrypt KX-S, get result $d = \alpha x\text{-}s + e$.
4)P1 outputs m1= s1+d, other Pi output mi= si.
Now the mi sum to $\alpha x + e$, where e is an error chosen by the adversary.

Note that the error is NOT a problem: we already showed the MAC is secure even if the adversary can add an error to the MAC. It doesn't matter that the error is added already in the preprocessing!

How to detect errors in multiplication triples.

We can now generate triples in correct format: [a], [b], [c], where we are not sure that c=ab, however.
Using similar method, can also generate representations of form [[t]].

To detect errors, we will do the following test for lots of triples in parallel, show only one here for simplicity.

Take another (also unreliable) triple [x], [y], [z] and do:

1)Open a representation of a random [[t]] (use same t for all triples tested)
2)Partially open t[a]-[x] to get u, and [b] –[y] to get v.
3)Partially open t[c] – [z] – v[x] – u[y] – uv. If result is not 0, abort.
4)Check MACs for all the partial openings, same protocol as in on-line phase.

Easy to show that if ab is not c, then test only goes through for ONE single value of t, or if MAC check fails. Both happen with probability 1/p.

Zero-Knowledge Proofs of Plaintext Knowledge.

Use classic design principle:

To show that cipertext C is well formed and he knows plaintext:
1)the prover P makes an aux. random ciphertext A.
2)A random challenge bit b is chosen by the verifier V
3)If $b=0$, open A (reveal plaintext and randomness), if $b=1$, open $C+A$.

Standard techniques can be used to choose a challenge that all players trust is random, so P does not have to do a proof to all players.

If A is chosen with proper distribution (details under the rug!), opening $C+A$ reveal nothing about C, so proof is ZK.

By homomorphic property, if P can open A and $C+A$, can also open
$C+A -A = C$
So soundness error is $\frac{1}{2}$. Too inefficient to just repeat this, but we can use amortization technique by [Damgård and Cramer, Crypto 09] to do many proofs in parallel more efficiently.

Proof of Security for Preprocessing Phase.

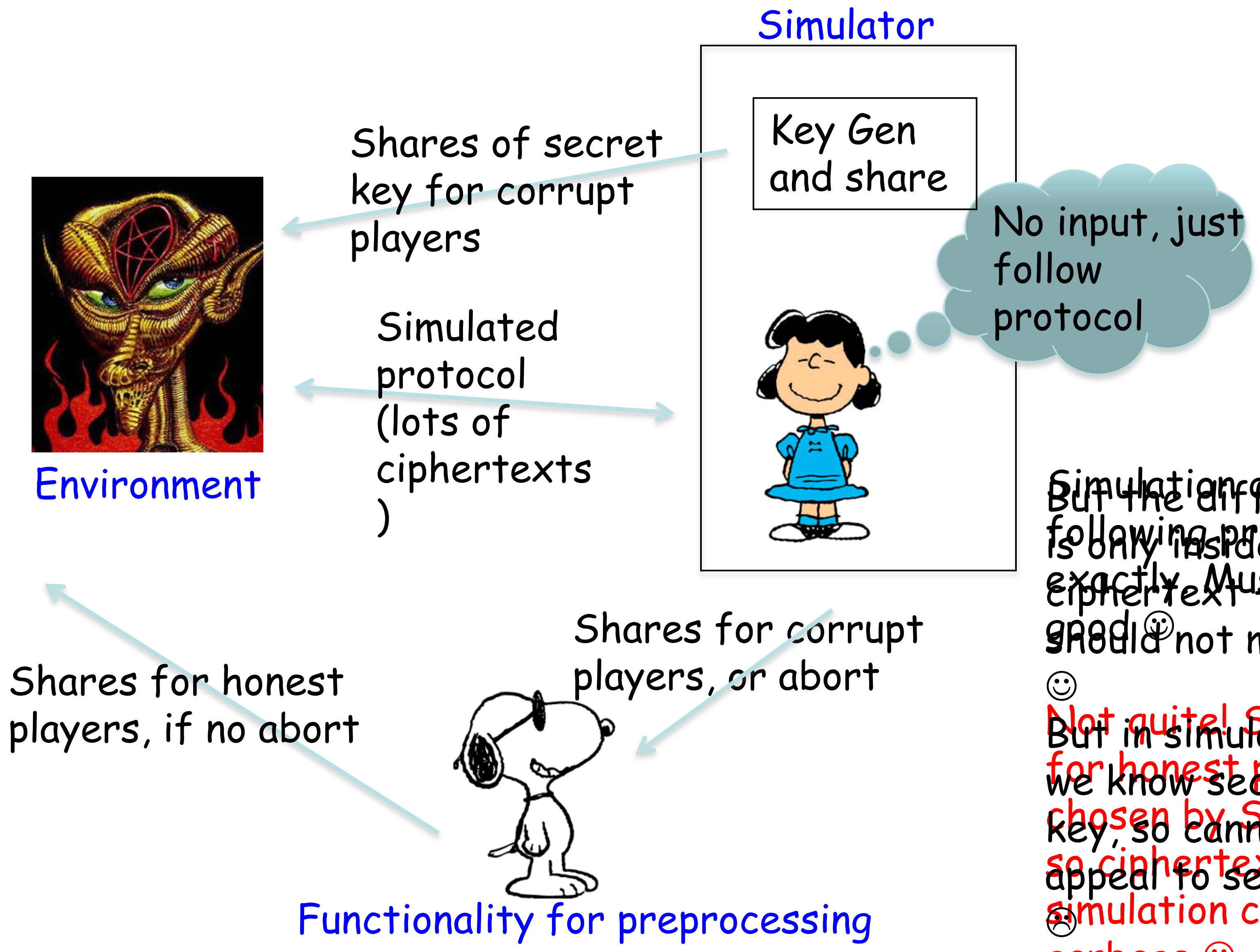
Preprocessing Functionality F (simplified)
Makes lots of representations [a], where

1) It gets from the adversary (simulator) the shares that corrupt players should have.
2) Then it chooses a at random and shares for honest players at random such that sum is correct.

Necessary that adversary chooses shares for corrupt players, else functionality cannot be implemented: not clear that a simulator could make protocol transcript match the choices of the functionality.

Simulator S

1) Generate a public and secret key, makes shares of secret key and send corrupt players their shares.
2) Now simulate by simply following the protocol. Decrypt ciphertexts from corrupted players to get their shares, send to F (note legal UC simulator, as no rewinding required).


Simulator
Key Gen and share
No input, just follow protocol
Shares of secret key for corrupt players
Simulated protocol (lots of ciphertexts)
Environment
Shares for corrupt players, or abort
Shares for honest players, if no abort
Functionality for preprocessing

Simulation done by following protocol exactly. Must be good ☺

But the difference is only inside ciphertext - should not matter ☺

But in simulation, we know secret key, so cannot appeal to security ☹

Not quite! Shares for honest players chosen by Snoopy, so ciphertexts in simulation contain garbage ☹

Proof of Security for Preprocessing Phase,part 2

Environment sees the shares of [a] that honest players output, as well the ciphertexts they send in the protocol

Real process
The ciphertexts contain the actual shares of honest players.
Ideal process
The ciphertexts contain unrelated values chosen by the simulator

Intuition: this should not matter, if the cryptosystem is CPA secure.

Idea for Reduction to show this (simplified)
1)Get a public key, and challenge ciphertexts with either real or random values.
2)Now do the same as the simulator S, except that we

1) use challenge ciphertexts to play the role of honest player's ciphertexts. Use ZK simulator to do the proofs for honest players.
2) Use rewinding to extract plaintexts of corrupt players' ciphertexts (legal, we are not doing UC simulation here!)

Implementation Results for Preprocessing

Number of public-key operations spent to prepare a secure multiplication is $O(n^2/s)$ where s is number of fields elements packed in one ciphertext (about 12000 in our implementation).

Implementation: 3 players connected on a LAN, security similar to 1024 bit RSA, spends 10-13 ms. (amortized time) to prepare for 1 secure 64-bit multiplication for 3 players.
Covert security with 10% probability to cheat: about 5 ms.

The overhead for small fields

On-line phase only “optimal” for fields where 1/field size is comparable to 2-k, where k is security parameter.

For small fields, e.g., F_2 overhead is a factor k.

Can we do better?

Doing Boolean Circuits with (almost) Constant Overhead, the "MiniMac" Protocol

[Damgård, Zakarias 2012]

Idea: try to implement the Boolean circuit by doing only blockwise operations, i.e.,
(b1,…,bk) + (v1,…,vk) = (b1 + v1,…,bk + vk)
(b1,…,bk) * (v1,…,vk) = (b1 * v1,…,bk * vk)

Hope: authenticating shares of a block should be cheaper than authenticating each bit individually.

Result from [DIK10]: can restructure any Boolean circuit so that only block-operations and a small number of permutations inside blocks are needed. Comes at cost a logarithmic blow-up.

A new Authentication Scheme for k-bit blocks

Secret value $x= x1+x2$, $x1,x2$ shares in x, + is now XOR
NEW: x must be in an error correcting code C.
MAC $m(x) = \alpha * x = m(x)_1 + m(x)_2$ * is bit-wise product
In addition dealer will issue shares $\alpha 1$, $\alpha 2$ to players such that $\alpha = \alpha 1 + \alpha 2$.

$x1$
$m(x)_1$

$x2$
$m(x)_2$

When opening, we check the MAC and that x is a codeword in C.
To cheat, must guess many entries in α, where many = min.dist. of C
Overhead of this MAC is constant if C has constant rate.

New Message Authentication Codes cnt'd.

[x] = (x1, m(x1) x2, m(x2))

Problem! multiplications by constants do not work the same way as before: say we have
mac(x) = α*x mult. by e on both sides, get e*mac$_K$(x) = α*(e*x)
Even if e is a codeword, no guarantee that e*a is a codeword!
However..

Solving the Problem using the Schur Transform

C*= Schur transform of C
Defined as linear span of all words of form x*y where x,y are in C.

e*a is in C*, so if also C* has large minimum distance, then we're fine.

e*mac(x) = α*(e*x) = mac(e*a)

to open, Lucy checks that e*a is in C*

Now, everything works as before. So we just need a family of codes C such that:

C and C* have constant information and error rates
C should be over a small field, ideally constant size.

We need to checking membership in C, C* efficiently
(would also like efficient encoding into C)

The Problem of Checking Membership in C

Naïve approach: multiply by parity check matrix H
Works, but work per data bit is order k. Much too expensive.

But we need to check many code words, so collect them in matrix A and check that $HA = 0$
Can now use smart matrix algorithms, work per data bit now k^d, d<1.
But we want constant!

Idea: let G be generator matrix for linear time encodable code, with constant error and data rates (e.g., Spielman – note: this is a different code from C).

May as well check that $G(HA)G^{\dagger} = 0$

The Problem of Checking Membership in C, cont.

May as well check that $G(HA)G^{\dagger} = 0$

If HA is not 0, then $G(HA)G^{\dagger}$ contains a costant fraction of 1's because G generates a code with large minimum distance.

Note: can compute GH and $AG^{\dagger}$ in quadratic time by linear time encodability

Can sample an entry in $G(HA)G^{\dagger}$ in linear time: take random row from GH and random column from $AG^{\dagger}$ and compute inner product.

Sample linear number of entries and check for 0.
Exponentially small error probability in quadratic total time and hence constant work per data bit!

Actually a general algorithm for verifying matrix products in quadratic time over small fields with exponentially small error prob. (co-invented with Yuval Ishai). Previous best was Freiwalds, error prob. 1/field size.

A final issue: Moving data around inside blocks.

Let's solve a more general problem: given [**x**], compute [f(**x**)], where f is a linear function.

Let preprocessing supply [**r**], [f(**r**)], for random **r**.

Compute and open [**x**] –[**r**] = [**x-r**] to get **x-r**.
All players compute f(**x-r**)
Output f(**x-r**) + [f(**r**)] = [f(**x**)].

When f just permutes entries, not too expensive that all compute f.

Works just as well for inputs that span across several blocks.

Circuit known in advance: can preprocess permutation that occur between layers.
Circuit not known in advance: put Benes network between layers a la [DIK10]. Costs a log(circuit size) factor, but now just need lots of permutations of single n-bit blocks. Only log(n) different permutations will occur.

Choice of Codes and Results

All results work for well-formed circuits, circuits with a “not too strange” structure. Circuit size S.
DEF: Overhead H, if we have to use O(HS) computational work, communcation and storage.

Can use Reed-Solomon codes based on GF(2^(log n)),
For instance F_{256} will work nicely in practice

Cannot get constant overhead as in SPDZ for large fields, but we do get polylog(k) log(S)
overhead.

More Results

Can use codes from Algebraic Geometry (Cramer et al.) then field size can be constant.

Hence get log(S) data and communication overhead, and O(1+k/n)log(S) computation overhead.

We get the O(1+k/n) factor and not O(1) because encoding in AG codes is not efficient, but we can let the players share the encoding work..

All log(S) factors disappear if circuit known in advance.

It's even easier if we want to compute the same function k times in parallel. Can use "bit-slicing". Then no need at all for rearranging inside blocks.

Implementation Results

In [DTZ, SCN13] did practical study of using MiniMac to compute many AES circuits in parallel.

Used Reed-Solomon codes based on F_{256} and an FFT variant to speed up encoding in C.

Constructed further optimization to use all 8 bits in each data byte.

Achieved about 3 ms, amortized, per AES operation.

Many ways to speed this up further for AES
– preprocess S Boxes
- preprocess linear mapping between SubBytes operations.
Future/Ongoing work..

Implementation of Preprocessing for MiniMac?

Very recent Work from Bristol:

Used TinyOT preprocessing to get preprocessed data for both SPDZ and MiniMac.

Open Problems and upcoming results.

Must we communicate for every mult. gate?

In honest majority setting: convert secret sharings of a and b into one of ab requires communcation, if the new sharing of ab has same threshold as before. Otherwise, can make 2-party secure computation of AND.

But this argument fails in the preprocessing model!

Upcoming work with Jesper Nielsen and Antigoni Polychroniadou:
Alice has a, Bob has b. Achieving an additive sharing of ab (as in SPDZ) is impossible in the preprocessing model without communication.
If the goal is a more general secret sharing of an output f(a,b), this impossible without communication if f() is "complex enough".

Round complexity: open if we can get constant number of rounds and information theoretic security efficiently. Even in preprocessing model. Above result indicates there is nothing to do if we go by the gate-by-gate approach. But in general??

Garbling Schemes or Protocols based on Arithmetic Circuits?

It depends! - on the computation we want to do.

If the goal is, e.g., linear programming (very relevent for benchmarking applications), want arithmetic on large integers. With SPDZ and the like, mult and add involving many bits is one gate..

If circuit has low depth, SPDZ/TinyOT will win in many cases (think sugar beets ☺).

Sometimes we want also operations that are naturally binary (comparison), but then let's find out how to convert formats efficiently and get the best of both worlds!

What about hardware support for SPDZ like protocols? Not much has been done..
Go for it! The sky is the limit! ☺

Thanks!