A Strategy for Provably Secure Multi-party Computation

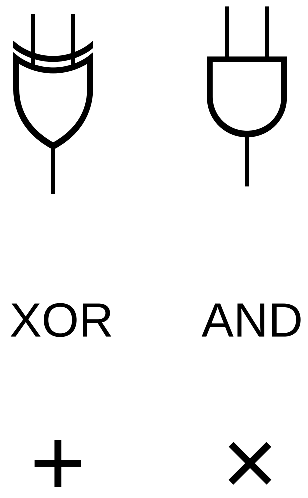
- What is MPC and why do we want it?
- How does MPC work?
- What does security mean *vis-à-vis* MPC?
- How do we know an MPC protocol is secure?
- A lengthy detour into modern patterns in functional programming.
- Can we use those patterns for MPC protocols and proofs?

What is MPC and why do we want it?

- We want to compute a function that takes secret inputs from each of us.
- *e.g.* We want to know how many doughnuts we should make, **without** telling each other how many doughnuts we *plan* to eat.
- Various configurations are possible,
  e.g. Homomorphic encryption covers a situation with two parties,
  one of whom has all of the secret inputs and relatively little
  computational power.

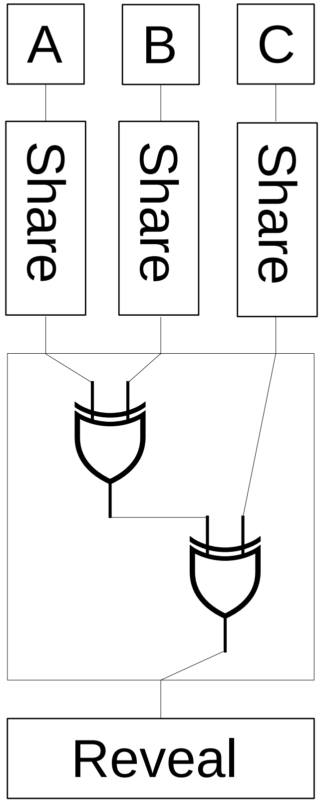
How does MPC work?

- Most protocols act on arithmetic expressions, *aka* **“circuits”**.
- *In theory*, any computable function can be expressed as an expression in terms of “+” and “×” on the inputs.
- If we use the finite field of size two, *aka* binary, then this corresponds to a circuit made of “XOR” and “AND” gates.

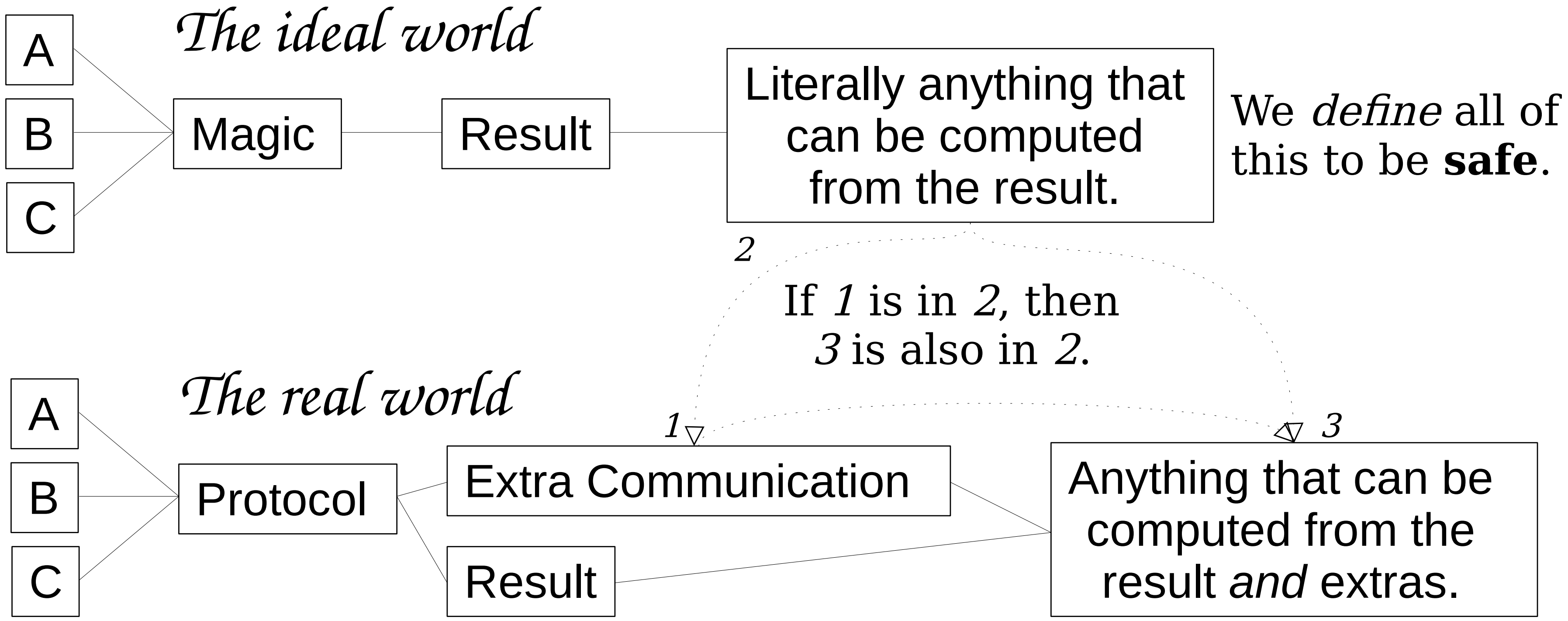


How does MPC work?

- A, B, and C each have a single secret bit. They want to learn the sum (XOR) of the three bits.
- **Share**: Send the other two parties random bits as their “shares” of you secret. Your own share of your secret is the XOR of your secret and the the other shares.
- **Compute** the circuit using normal arithmetic on your shares.
- **Reveal:** Everyone sends everyone their shares of the final value. XOR these to get the answer.



What does “secure” mean for an MPC system?



How do we know if a particular system is secure?

- Proving *a* particular system to be secure isn’t too hard.
- The traditional systems all have limitations or caveats that limit the situations in which they’re useful. This includes major performance issues (bandwidth, sequential rounds of communication, compute-power).
- All the additional tricks and complications we add to improve performance or extend a protocol to other situations require their own proofs of security.
- Two individually secure systems *might* be safe to use in conjunction/composition, *depending on details*.

How do we know if a particular system is secure?

- We **would like** a framework for saying
  ”This implementation of X in terms of Y is safe *assuming* Y gets implemented safely.”

- We **would like** a framework that could check safety *automatically* and *statically*.

And now for something completely different!

A **free monad** is the name for a way of writing a program in a kinda DSL, which can have one or more implementations defined elsewhere.

Extensible effects systems let you mix'n'match operators from disparate signature declarations to build the basis of your free monad.

The "State" API:

```
get<a> :: ()→M(Δ)[a]
set<a> :: a→M(Δ)[()]
  where State<a> ∈ Δ
```

The "Log" API:

```
log :: String→M(Δ)[()]
  where Log ∈ Δ
```

A program with $\Delta = \{State,\ Log\}$*:*

```
do {
  log("update state")
  set("foobar") }
```

Algebraic effects:

- Free-monad extensible effects are basically the same as “free algebras” from category theory.*

  *ignore this

- Algebraic effects are “algebraic” because they bring back into programming-practice the “equational” attributes that are part of a free algebra’s declaration.**

  **roughly speaking

- An implementation of an algebraic-effects API is only “correct” if it preserves the API’s equations.

Algebraic effects:

The "State" API:

```
State<A>:
    get<a> :: ()→M(Δ)[a]
    set<a> :: a→M(Δ)[()]
        where State<a> ∈ Δ

    do{ set(x) ; get() } == do{ set(x) ; return x }

    do{ set(x) ; set(y) } == do{ set(y) }

    do{ a ← get() ; get() } == do{ get() }
```

Can we use algebraic effects for MPC proofs?

AE systems allow a mix’n’match approach to both APIs and implementations of APIs, and enforce properties of those APIs! *but…*

- The correctness of an AE API implementation is not computationally decidable without additional limitations!
- Few “real” AE systems exist!
- The fundamental property we want to prove about MPC systems isn’t even equational!

A path forward:

- Figure out a suitable proxy for MPC security that can be checked by static analysis.
- Develop a framework for asserting and tracking non-algebraic properties of free-monad APIs.
- Develop a framework for representing and (with limitations) automatically deriving composable proofs for the above system

Thank you professors Joe Near
& Chris Skalka!

For more on MPC systems and proofs, consider
A Pragmatic Introduction to Secure Multi-Party Computation
Evans, Kolesnikov, & Rosulek; 2018/2020

For more on algebraic effects *theory*, try
An Introduction to Algebraic Effects and Handlers
Pretnar; 2015

For extensible effects as used in industry, consider
the `Polysemy` library for Haskell.