Ongoing work:
Verifying the safety of a MPC/SMC protocol & language by Coq

Greg Weng TPP 2023

Greg Weng

2023.Feb: Mercari R4D program -> Nagoya University Ph.D. student (大学院多元数理科学研究科)

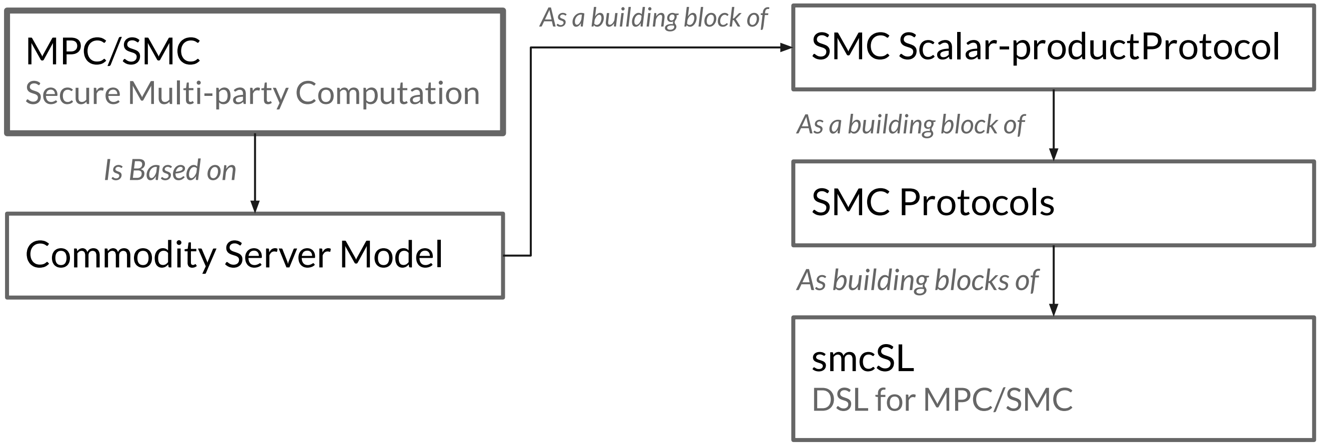
2019 ~: Mercari, software engineer: *Golang*

2017 - 2019: Rakuten, software engineer: *JavaScript*

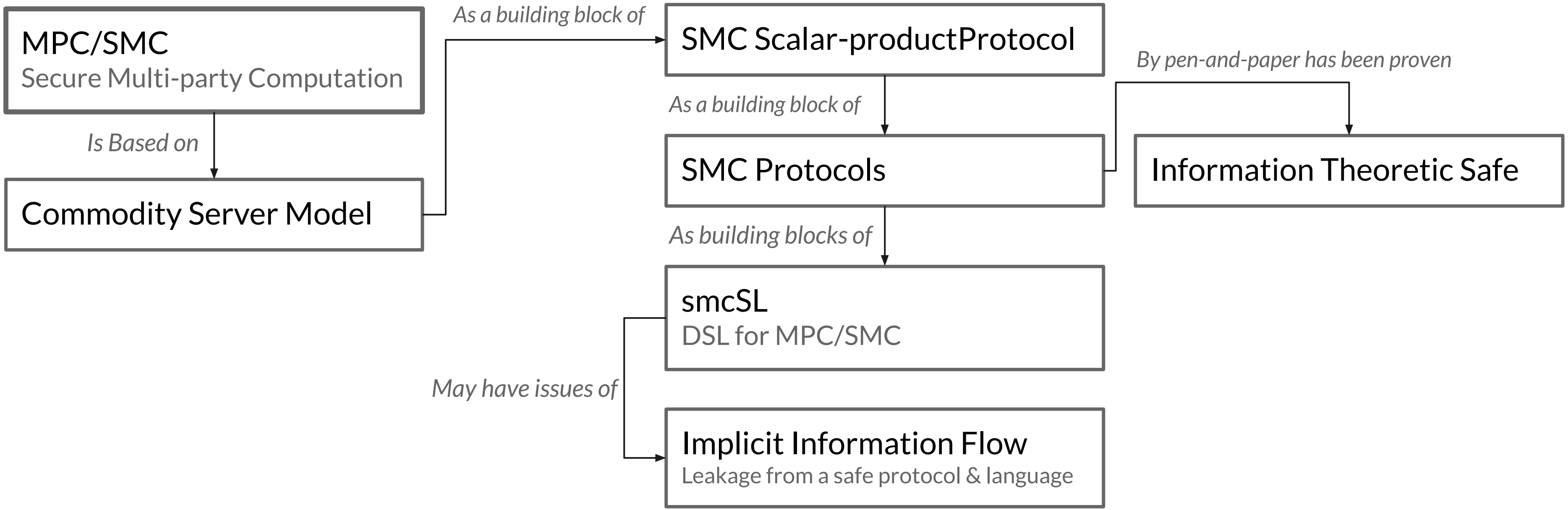
2014 - 2017: Mozilla Taipei, software engineer: *C, C++, JavaScript*

2012 - 2014: National Chengchi University (Taiwan), CS department (master's degree): *Haskell, Ruby*

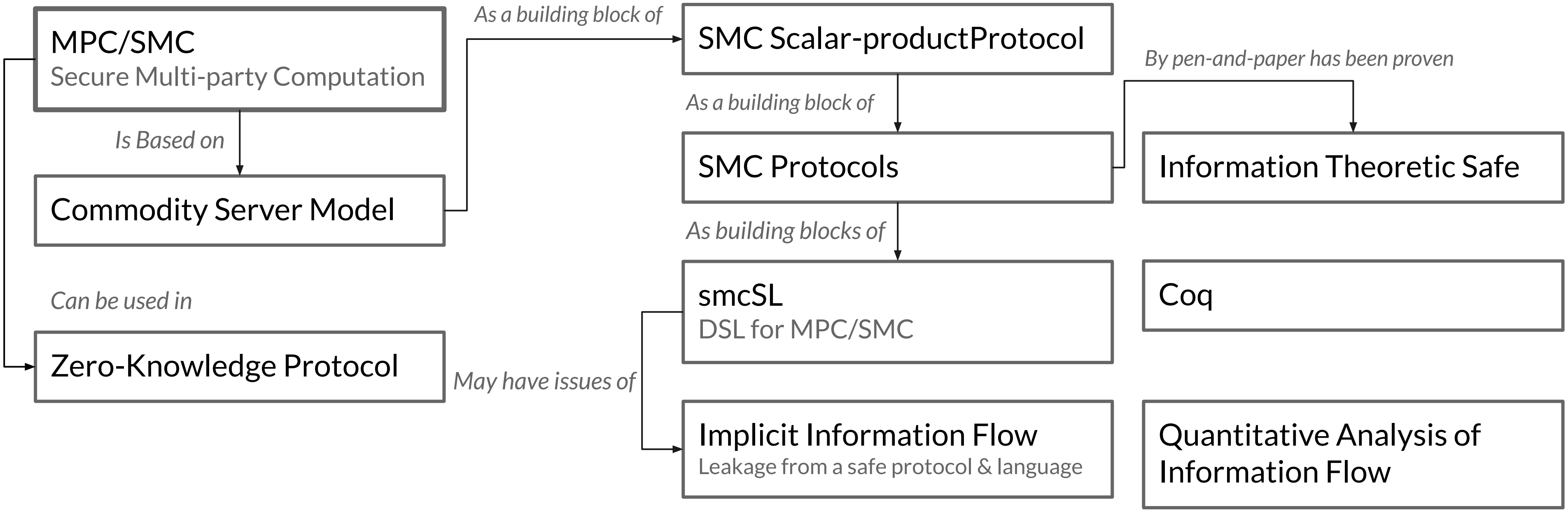
Research Background



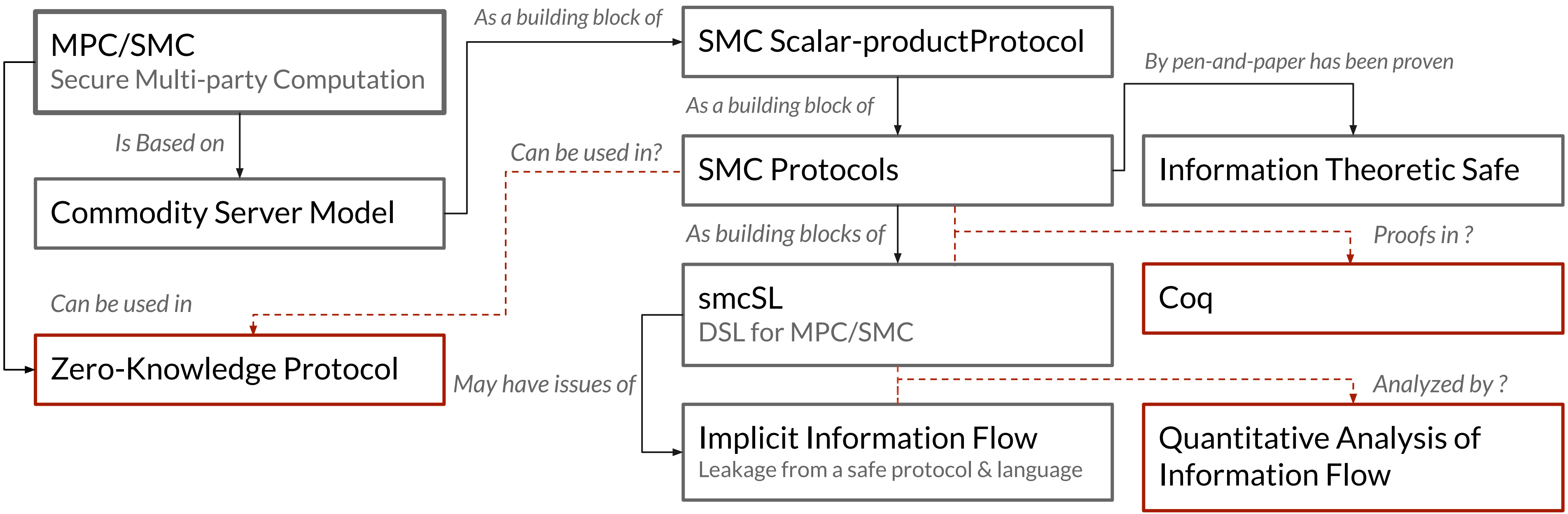
Research Background



Research Background



Research Background



MPC/SMC

MPC/SMC stands for "Multi-Party Computation" and "Secure Multi-party Computation"

It is a security domain about two or more parties collaboratively compute results, without revealing each party's secret data over what they agree to share.

Nowadays this tech is also used for digital wallet

Ex: online auction, voting, medical data analysis

Neither A nor B agree to reveal their bidding price
Both of them want to know the bidding result (A >? B)
--> This indirectly reveal other properties of their secrets:

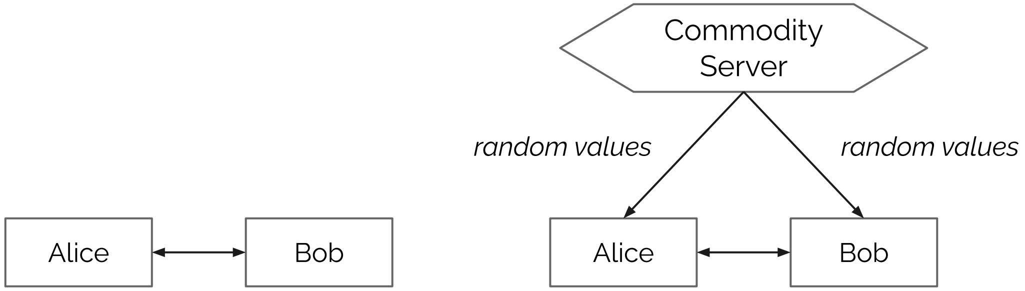
If not (A > B),
for A, a new fact is that B's price is larger than $500
for B, a new fact is that A's price is smaller than $900

Commodity Server Model MPC/SMC

It is a model proposed by Wenliang Du and Zhijun Zhan*, to simplify the infrastructure and computation difficulties for two and more parties, by introducing a "commodity server" in the MPC/SMC computation flow.

The only role of this commodity server, is to issue necessary random values to guarantee when Alice and Bob compute collaboratively

With MPC/SMC protocols, no one should obtain more information than they agree to share, from the data they pass to each other.



Two-Party Model

Commodity-Server Model

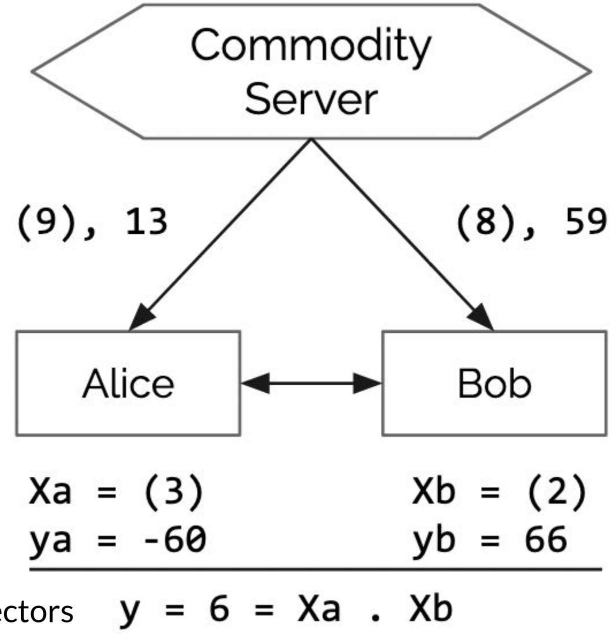
* Wenliang Du, Zhijun Zhan. A practical approach to solve Secure Multi-party Computation problems. NSPW 2002: Proceedings of the 2002 Workshop on New Security Paradigms; 2002 Sep 23-26; Virginia Beach, Virginia USA. New York, NY, USA: ACM Press; 2002. p. 127-35.

Scalar-product based MPC/SMC protocols

[Scalar product protocol -- Commodity server approach] ref1

```
For example:

Xa = (3), Xb = (2)
Commodity Ra, Rb, ra, rb = (9), (8), 13, 59
Results ya, yb = -60, 66

ya + yb = 6 = (3) . (2) = Xa . Xb
```

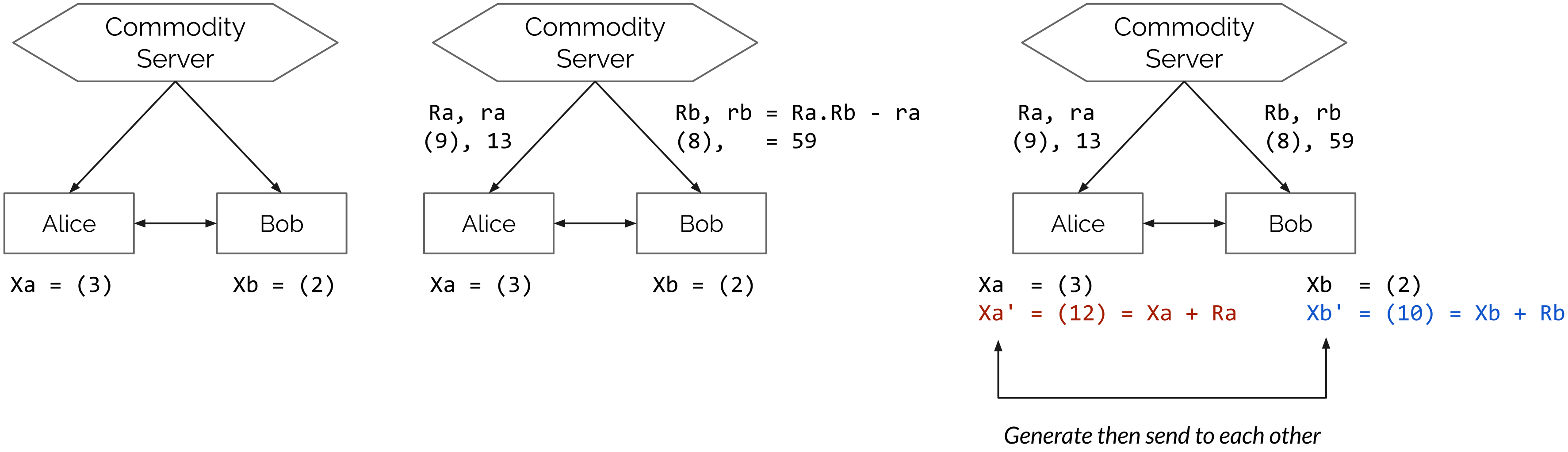
Local inputs: Xa, Xb
Shared output: ya, yb



→ Alice and Bob collaboratively computed the result y = ya + yb, where y = Xa . Xb
→ If numbers are real numbers*, Alice and Bob cannot know each other's secret vectors

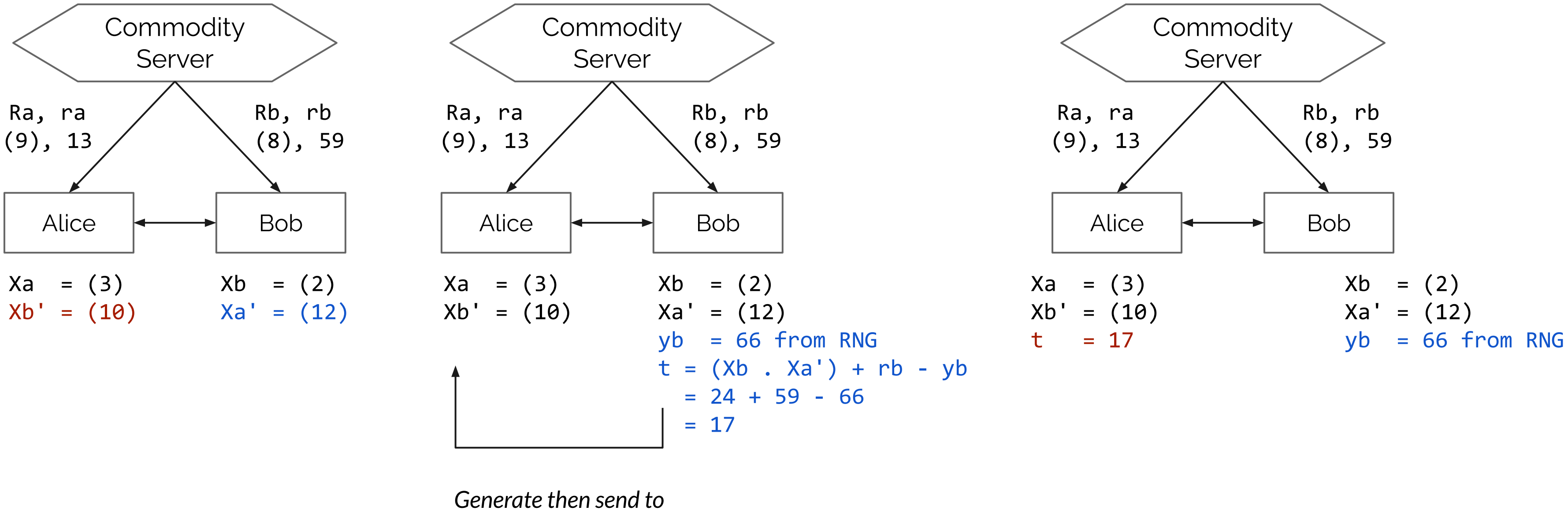
Scalar-product based MPC/SMC protocols

[Scalar product protocol -- Commodity server approach] ref1



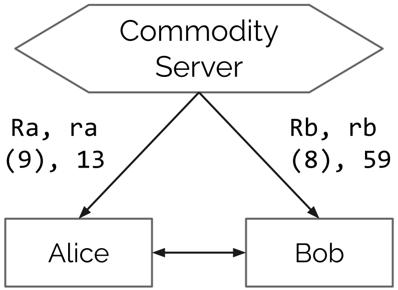
Scalar-product based MPC/SMC protocols

[Scalar product protocol -- Commodity server approach] ref1



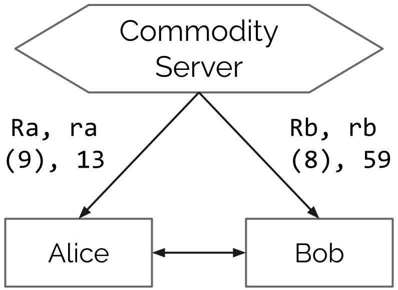
Scalar-product based MPC/SMC protocols

[Scalar product protocol -- Commodity server approach] ref1




```
Xa  = (3)
Xb' = (10)
t   = 17
ya  = t - (Ra . X'b) + ra
    = 17 - 90 + 13
    = -60
```

```
Xb  = (2)
Xa' = (12)
yb  = 66 from RNG
```




```
Xa  = (3)        Xb  = (2)
ya  = -60        yb  = 66
```

```
y = 6 = ya + yb = Xa . Xb
```

Scalar-product based MPC/SMC protocols

[Scalar product protocol -- Commodity server approach] [ref1]

(Alice, Bob) hold

Local inputs (Xa, Xb)

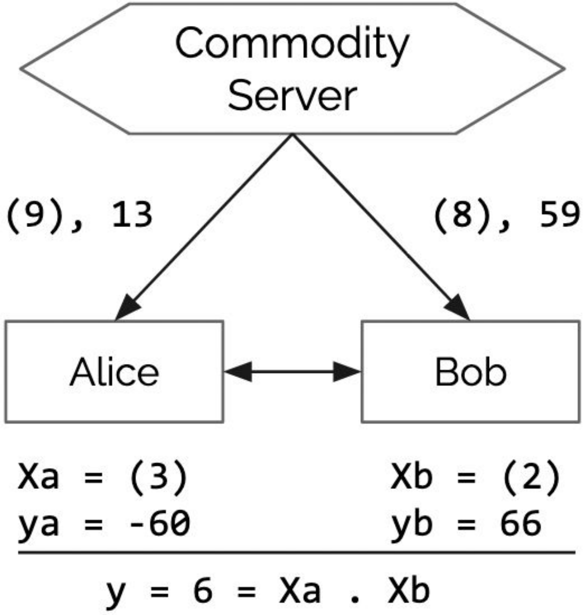
Shared outputs (ya, yb)

This Scalar-product protocol can be denoted as [ref2]:

$$((X[1]_a, ..., X[d]_a), (X[1]_b, ..., X[d]_b)) \rightarrow (y_a, y_b), \textit{where}$$

$$y_a + y_b = X_a \cdot X_b = \sum_{i=1}^{d} X[i]_a \cdot X[i]_b$$

where $X[i]_a, X[i]_b, y_a, y_b \in \mathbb{Z}$,
and $+$, $\cdot$ *are the modular additional and multiplication in* Z_n



Scalar-product based MPC/SMC protocols

[Scalar product protocol -- Commodity server approach] [ref2]

$$((X[1]_a, ..., X[d]_a), (X[1]_b, ..., X[d]_b)) \rightarrow (y_a, y_b), where$$

$$y_a + y_b = X_a \cdot X_b = \sum_{i=1}^{d} X[i]_a \cdot X[i]_b$$

$$where\ X[i]_a, X[i]_b, y_a, y_b \in \mathbb{Z},$$
$$and\ +\ ,\ \cdot\ are\ the\ modular\ additional\ and\ multiplication\ in\ Z_n$$

By this basic building block, Academia Sinica in Taiwan built other secure protocols for MPC/SMC arithmetic operations, including comparison, conditional expression, etc[ref2] .

In following research [ref3], more protocols were invented to support both integers and floating points.

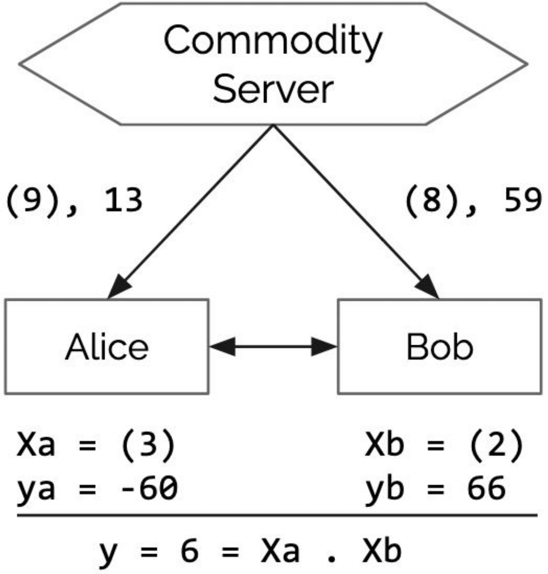
Scalar-product based MPC/SMC protocols[ref2]

[Scalar product protocol -- y = Xa . Xb]

$((X[1]_a, ..., X[d]_a), (X[1]_b, ..., X[d]_b)) \rightarrow (y_a, y_b), where$

$$y_a + y_b = X_a \cdot X_b = \sum_{i=1}^{d} X[i]_a \cdot X[i]_b$$

$where\ X[i]_a, X[i]_b, y_a, y_b \in \mathbb{Z},$
$and\ +\ ,\ \cdot\ are\ the\ modular\ additional\ and\ multiplication\ in\ Z_n$



Scalar-product based MPC/SMC protocols[ref2]

[Conditional expression -- Alice ba xa ya, Bob bb xb yb, result: (b ? xa + xb : ya + yb)]

1. A number b, a value if true x and a value if false y are shared by party a and b:

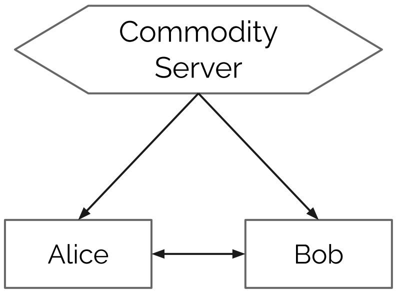
$$(b_a, x_a, y_a)$$
$$(b_b, x_b, y_b)$$

2. Party a and b joinly execute the protocol

$$((b_a, x_a, y_b), (b_b, x_b, y_b)) \to (z_a, z_b)$$

such that:

$$z_a + z_b = \begin{cases} x_a + x_b, & \text{if } b_a + b_b = 1 \\ y_a + y_b, & \text{if } b_a + b_b = 0 \end{cases}$$




```
Local Inputs:       (ba, xa, ya)     (bb, xb, yb)
Protocol Outputs:    za               zb
```

$$z = z_a + z_b = \begin{cases} x_a + x_b, & \text{if } b_a + b_b = 1 \\ y_a + y_b, & \text{if } b_a + b_b = 0 \end{cases}$$

Scalar-product based MPC/SMC protocols[ref2]

[Product -- Alice: xa ya, Bob: xb yb, result: z = x* y]

1. Numbers x and y are shared by party a and b:

$$(x_a, y_a)$$
$$(x_b, y_b)$$

2. Party a and b joinly execute the Scalar-product protocol with input vectors (x_a, y_a) and (x_b, y_b): $((x_a, y_a), (x_b, y_b) \to (t_a, t_b))$, such that:

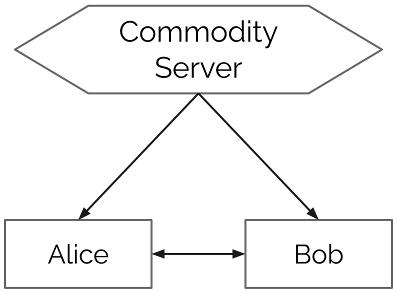
$$t_a + t_b = x_a y_b + y_a x_b$$

3. Party a and b locally compute:

$$z_a = t_a + x_a y_a$$
$$z_b = t_b + x_b y_b$$

4. The final results:

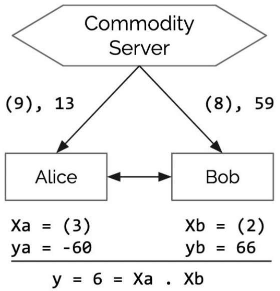
$$\begin{aligned} & z_a + z_b = \\ & x_a y_a + x_b y_b + (t_a + t_b) = \\ & x_a y_a + x_b y_b + (x_a y_b + y_a x_b) = \\ & (x_a + x_b) * (y_a + y_b) = x * y \end{aligned}$$



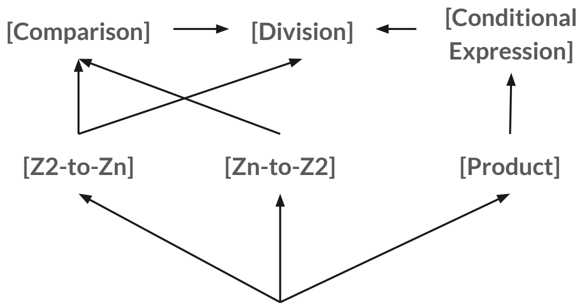

```
Local Inputs:       (xa, ya)        (xb, yb)
Protocol Outputs:    za              zb

z = za + zb = (xa + xb) * (ya + yb) = x * y
```

All protocols are from: Shen, Chih-Hao, Justin, Zhan, Tsan-Sheng, Hsu, Churn-Jung, Liau, and Da-Wei, Wang. "Scalar-product based secure two-party computation." . In 2008 IEEE International Conference on Granular Computing (pp. 556-561).2008.

Scalar-product, protocols, and the scripting language



[Scalar product protocol -- y = Xa . Xb, (commodity approach)] Ref.1



[Scalar product protocol -- y = Xa . Xb, (commodity approach)]

Arrow means: **A -(is used to build)->B** Ref.2

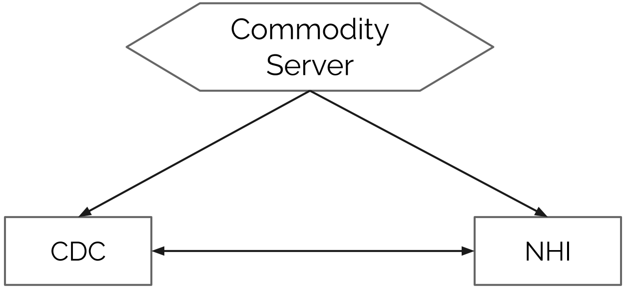
Compose SMC program in *smcSL*
To invoke SMC protocols **safely**

```
fun main( ): Shared Int
{
    Var asset_a: Alice Int ,
        asset_b: Bob Int ,
      richest: Shared Int ,
      n: Public Int

     richest = compare(asset_a, asset_b)  // Function call
     return  richest
}
```

From Weng, Cheng-Hui, Chen Kung, "A Modular Scripting Language for Secure Multi-Party Computation", master thesis

SMC Scalar-product —*is used to build*→ **SMC Protocols** —*is used to build*→ **SMC Scripting Language (smcSL)**

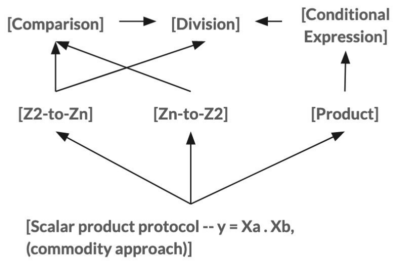
Application: Public health data analysis[ref8]



Chen, K., Hsu, T. S., Huang, W. K., Liau, C. J., & Wang, D. W. (2012). Towards a Scripting Language for Automating Secure Multiparty Computation.

```
CDC Local Inputs: Array of (0,1,0,0,.....1) (length: 23 millions; population in Taiwan )
Meaning: #N person got Dengue fever (a seasonal epidemic) = 1 or 0

NHI Local Inputs: Array of (0,1,0,0,.....1) (length: 23 millions; population in Taiwan )
Meaning: #N person in the past season receive outpatient or hospitality services due to Dengue fever

Result: Array of (1,0,0,0,.....0) (length: 23 millions)
Meaning: how much did the Dengue fever cost in this season

SMC Protocols used: conditional expression:

total := CDC[i] == NHI[i] ? total+1 : total
```

Research Goal: Verification in Coq (SMC protocols)



SMC Protocols

1. Verify SMC Scalar-product (require: list/vector libs) (GitHub repo: weng-chenghui/smc-coq)
2. Build protocols as ref.2 describes and prove their properties
3. Export them as module functions
4. Make protocols in the form `(Xa, Xb)-> (ya, yb)` Monadic
   a. `->` may be a SMC interface operation for different instances
   b. Find some laws like in the paper Just do it: simple monadic equational reasoning Ref5

Research Goal: Verification in Coq (smcSL)

Compose SMC program in *smcSL*
To invoke SMC protocols **safely**

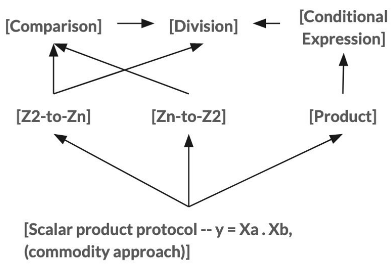
```
fun main( ): Shared Int
{
     Var asset_a: Alice Int ,
         asset_b: Bob Int ,
    richest: Shared Int ,
    n: Public Int

   richest = compare(asset_a, asset_b)  // Function call
   return  richest
}
```

SMC Scripting Language (smcSL)

Three types of monad: Local, Observation, Command (reference: monae)

1. Local: where actual states are manipulated
2. Observation: where traces for reasoning can be collected -- inputs and outputs to each local
3. Command: where smcSL program interpreted to SMC protocol executations
4. Formalize the information flow like in the paper: Quantitative information flow with monads in haskell [Ref6]

The original motivation: even if protocols have been proven safe [ref3], the information flow may still leak something



SMC Protocols

has been proven information-theoretically safe[ref3]

Compose SMC program in *smcSL*
To invoke SMC protocols **safely**

```
fun main( ): Shared Int
{
    Var asset_a: Alice Int ,
        asset_b: Bob Int ,
    richest: Shared Int ,
    n: Public Int

    richest = compare(asset_a, asset_b)  // Function call
    return  richest
}
```

SMC Scripting Language (smcSL)

can still leak some information *if* it builds the conditional loop as a language feature

→ implicit information flow
→ how to detect and how to quantify the leakage?

Just do it: simple monadic equational reasoning[Ref5]

Authors: Jeremy Gibbons and Ralf Hinze

It shows:

1. How to prove a program's claims by reasoning each monadic program step
2. These reasoning steps (and thus the proof) are instance-independant
    a. With only the monadic interface, one can claim and prove properties without knowing the instance

→ SMC protocols can be described in monadic steps, and thus their claims can be reasoned in the same way

→ Especially these protocols are actually used by the domain-specific language: smcSL

Quantitative information flow with monads in haskell[Ref6]

Authors: Jeremy Gibbons, Annabelle McIver, Carroll Morgan, Tom Schrijvers

It shows:

1. How to define a Monad with probability and combine it with the information leakage analysis
   a. Tracing how much information will be leaked in programs that are composed by leaking monadic operations
2. A language (Kuifje) and use it to analyze information leakage with state updating programs

→ A related work for analyzing smcSL

→ Yet the issue "protocols are safe but progam leaks information" still need some more work

Possible extensions

1. Extend the commodity-server-based SMC scalar product to N parties, not just two parties
    a. Instead of scalar product, ***determinant*** seems to have the potential to extend the protocol to N parties
    b. But need to solve the problem of padding numbers when inputs cannot form a square matrix
2. And also extend the SMC protocols build on it
3. With Coq verification
4. Use and extend this N parties SMC protocol to the zero-knowledge-protocol that described in the paper [ref7]:

    Zero-knowledge from secure multiparty computation

Zero-knowledge from secure multiparty computation[ref7]

Authors: Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Amit Sahai

It shows:

1. A N-party MPC/SMC program can be used as a problem in ZKP with lower cost compared to 3-coloring problem or Hamiltonicity
2. The zero-knowledge protocol use such a MPC/SMC program can satisfy three properties (completness, soundness, and zero-knowledge), even if there are some corrupted MPC/SMC players

Difficulties & finds

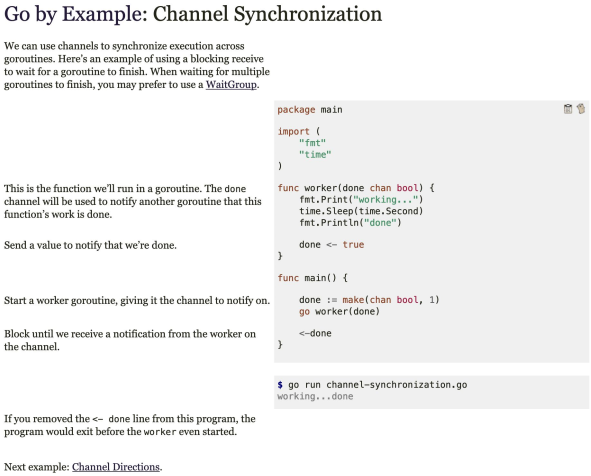
1. Coq
2. Work & Study at the same time
3. Culture shock: academic vs. industrial documents, codes, discussions, and strategies to solve problems

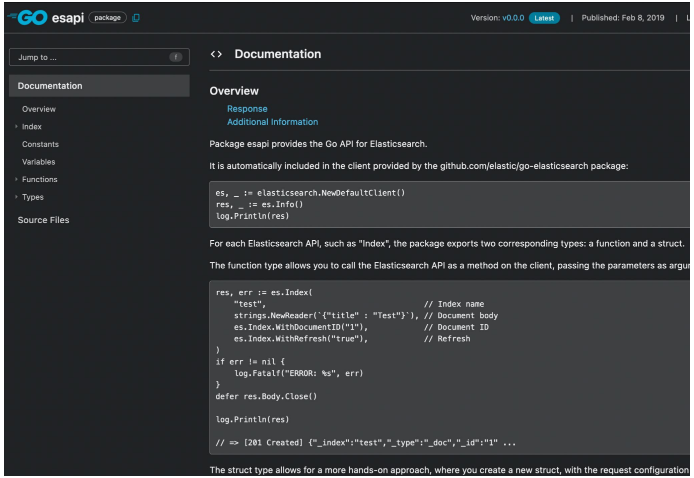
Difficulties & finds (Coq)

As a software engineer, most of time people expect to find immediately usable examples or specs.

Or at least with only code, it is still readable to understand how the code work.

Also with API references, one should be able to complete most of work without asking anyone.





Difficulties & finds (Coq)

As a Ph.D. student,

Most of time people expect to find immediately usable examples or specs.

Or at least with only code, it is still readable to understand how the code work.

Also with API references, one should be able to complete most of work without asking anyone.

→ I'm still trying to get used to Coq coding style (with SSReflect) ,

and how to accept myself when a single **//** blocks me to parse the whole proof,

or when there is a bug in old Coq code, by adding some magic terms or splitting steps the issues are gone

Difficulties & finds (culture shock)

As a software engineer, most of time we are asked to workaround issues & copy existing solutions to meet business requirements.

There is no time and no priority to study and solve a problem thoroughly, or create something with comprehensive design.

For solving problems

1. Can we reduce the impact to end users, or other systems?
2. Can we solve it within one week?
3. Is there any existing workaround or solution we can easily apply?
4. Does this problem with reduced impact, really worth to solve?

For building something new

1. Can we reach business requirements fast and cheap?
2. If generalization means no one can see its value, we choose copying code and specializing it for one business purpose.
3. "Edge cases" = never need to prevent them from happening, unless they really happen

Difficulties & finds (culture shock)

Also as a software engineer, people expect to output some results within reasonable time

Or if after a while we cannot have the expected result, at least we know the reason

Difficulties & finds (culture shock)

Also as a software engineer, people expect to output some results within reasonable time.

Or if after a while we cannot have the expected result, at least we know the reason.

And how much resource we already spent, and if we continue, how much we will spend.

Result <-> Time <-> People <-> Priority <-> Impact

Difficulties & finds (culture shock)

Also as a software engineer, people expect to output some results within reasonable time.

Or if after a while we cannot have the expected result, at least we know the reason.

And how much resource we already spent, and if we continue, how much we will spend.

Result <-> Time <-> People <-> Priority <-> Impact

As a Ph.D. student, there are so many possible paths, papers, ideas, things-to-study that may all lead to a dead end.

Result <?> Time <?> People <?> Priority <?> Impact

Q&A

References

1. Wenliang Du, Zhijun Zhan. A practical approach to solve Secure Multi-party Computation problems. NSPW 2002: Proceedings of the 2002 Workshop on New Security Paradigms; 2002 Sep 23-26; Virginia Beach, Virginia USA. New York, NY, USA: ACM Press; 2002. p. 127-35.

2. Shen, Chih-Hao, Justin, Zhan, Tsan-Sheng, Hsu, Churn-Jung, Liau, and Da-Wei, Wang. "Scalar-product based secure two-party computation." In 2008 IEEE International Conference on Granular Computing (pp. 556-561).2008.

3. Wang, D. W., Liau, C. J., Chiang, Y. T., & Hsu, T. S. (2006). Information theoretical analysis of two-party secret computation. In Data and Applications Security XX: 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sophia Antipolis, France, July 31-August 2, 2006. Proceedings 20 (pp. 310-317). Springer Berlin Heidelberg.

4. Weng, Cheng-Hui, Chen Kung "模組化之安全多方計算領域專屬腳本語言 A Modular Scripting Language for Secure Multi-Party Computation"

5. Gibbons, J., & Hinze, R. (2011). Just do it: simple monadic equational reasoning. ACM SIGPLAN Notices, 46(9), 2-14.

6. Gibbons, J., McIver, A., Morgan, C., & Schrijvers, T. (2019). Quantitative information flow with monads in haskell. Foundations of Probabilistic Programming.

7. Ishai, Y., Kushilevitz, E., Ostrovsky, R., & Sahai, A. (2007, June). Zero-knowledge from secure multiparty computation. In Proceedings of the thirty-ninth annual ACM symposium on Theory of computing (pp. 21-30).

8. Chen, K., Hsu, T. S., Huang, W. K., Liau, C. J., & Wang, D. W. (2012). Towards a Scripting Language for Automating Secure Multiparty Computation.