gpsMPC

GPS tools combined with Multi-Party Computation

Fabian Gruber

December 12, 2024

Outline

Introduction

Motivation

- Rising privacy concerns increase importance of MPC
- High complexity and performance costs slow down adoption
- Integration should be faster and easier

Fabian Gruber

Goal

- Map RISC-V instructions to MPC operations
- Virtual Machine (VM) abstracts MPC away
- Base 64-bit instruction set + Important extensions
- Users only have to care about inputs

Background

- Cryptographic protocols that allow secret computation
- Inputs can be secret or public
- Parties learn only results
- Operate on shares (Additive, Shamir [Sha79])

- Different protocols for different numbers of parties
- Security assumptions
  - Semi-honest/malicious adversary
  - Honest/dishonest majority
- Used for Private Set Intersection, Statistics, Privacy-Preserving Machine Learning

- Mixed protocols allow efficient computation
- E.g. ABY uses (A)rithmentic, (B)inary and (Y)ao shares [DSZ15]
- We implemented A2B and B2A conversions
  - Arithmetic share: $[x]$
  - Binary share: $\langle x \rangle$

- (R)educed (I)nstruction (S)et (C)omputer
- (C)omplex (I)nstruction (S)et (C)omputer
- Less and simpler vs. more and complicated instructions

RISC-V

- Free and open-source RISC architecture
- Base integer ISAs RV32I and RV64I
- 31 general-purpose registers `x1-31`
- Additional instruction extensions

Extensions

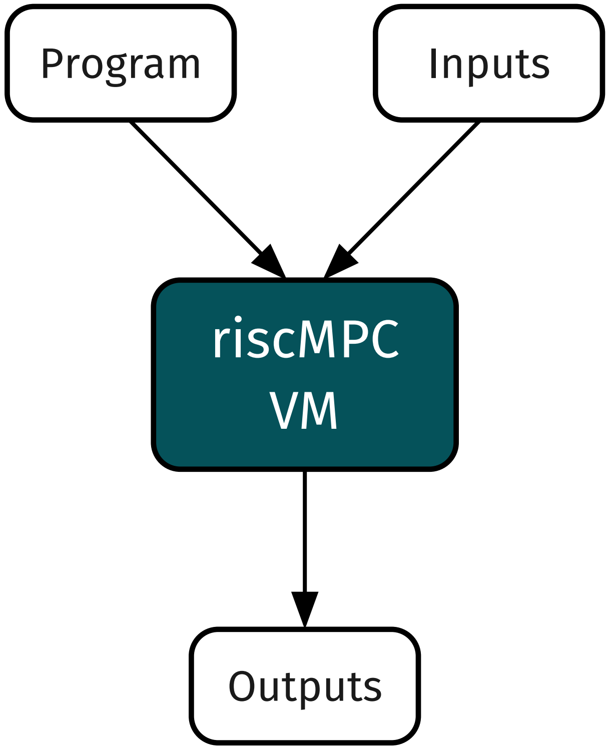
Extension	Description
"M"	Integer Multiplication and Division
"A"	Atomic Instructions
"F"	Single-Precision Floating-Point
"D"	Double-Precision Floating-Point
"Q"	Quad-Precision Floating-Point
"L"	Decimal Floating-Point
"C"	Compressed Instructions
"B"	Bit Manipulation
"J"	Dynamically Translated Languages
"T"	Transactional Memory
"P"	Packed-SIMD Instructions
"V"	Vector Operations

Registers

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5`	`t0`	Temporary/alternate link register	Caller
`x6`–`7`	`t1`–`2`	Temporaries	Caller
`x8`	`s0`/`fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10`–`11`	`a0`–`1`	Function arguments/return values	Caller
`x12`–`17`	`a2`–`7`	Function arguments	Caller
`x18`–`27`	`s2`–`11`	Saved registers	Callee
`x28`–`31`	`t3`–`6`	Temporaries	Caller
`f0`–`7`	`ft0`–`7`	FP temporaries	Caller
`f8`–`9`	`fs0`–`1`	FP saved registers	Callee
`f10`–`11`	`fa0`–`1`	FP arguments/return values	Caller
`f12`–`17`	`fa2`–`7`	FP arguments	Caller
`f18`–`27`	`fs2`–`11`	FP saved registers	Callee
`f28`–`31`	`ft8`–`11`	FP temporaries	Caller

riscMPC

- riscMPC implements a 2-Party semi-honest MPC setting
- Both parties execute RISC-V assembly instructions
- Arguments can be secret or public



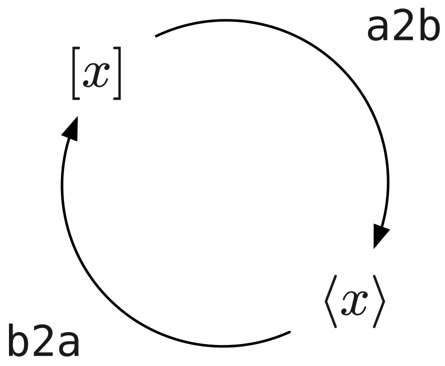
riscMPC Virtual Machine

Features	Project	Thesis
RV64IM (base 64-bit instructions + mul/div)	✓	✓
Double precision floating-point	×	✓
Vector operations	×	✓*
Fast offline phase (OT extension)	×	✓

Data Types

- Share enum represents arithmetic or binary shares

```rust
enum Share {
  Arithmetic(u64),
  Binary(u64),
}
```



- Integer enum represents secret or public integers

```
enum Integer {
  Secret(Share),
  Public(u64),
}
```

- Float enum represents secret or public floating-point numbers

```
enum Float {
  Secret(u64),
  Public(f64),
}
```

Instruction Examples

Instruction	Description
`add rd, rs1, rs2`	Add `rs1` and `rs2` store in `rd`
`xor rd, rs1, rs2`	Xor `rs1` and `rs2` store in `rd`
`mul rd, rs1, rs2`	Multiply `rs1` and `rs2` store in `rd`
`blt rs1, rs2, label`	Branch to `label` if `rs1 < rs2`
`fsqrt rd, rs1`	Square root of `rs1` store in `rd`

Addition

- Parties locally compute $[\mathrm{rs1}] + [\mathrm{rs2}]$
- Public operand 1 party computes $[\mathrm{rs1}] + \mathrm{rs2}$

Comm	Setup
-	-

Binary XOR

- Parties locally compute $\langle \mathrm{rs1} \rangle \oplus \langle \mathrm{rs2} \rangle$
- Public operand 1 party computes $\langle \mathrm{rs1} \rangle \oplus \mathrm{rs2}$

Comm	Setup
-	-

Multiplication

- Naive approach fails
- Instead use Mult. Triple $([a], [b], [c])$ [Bea92]

$$
\begin{aligned}
[d] &= [\mathrm{rs1}] - [a] \\
[e] &= [\mathrm{rs2}] - [b] \\
[\mathrm{res}] &= [c] + d[\mathrm{rs2}] + e[\mathrm{rs1}] + de
\end{aligned}
$$

- Public operand all parties computes $\mathrm{rs2}\ [\mathrm{rs1}]$

Comm	Setup
2 $\times$ 64-bit	1 MT

Branch if Less-Than

- Secretly compute sign-bit:

$$[s] = [\mathrm{rs1}] - [\mathrm{rs2}]$$
$$\langle s \rangle = \mathrm{A2B}([s])$$
$$\langle \text{sign-bit} \rangle = \langle s \rangle \gg 63$$

- Open sign-bit, take branch if it's 1 (negative difference $\rightarrow \mathrm{rs1} < \mathrm{rs2}$)
- A2B conversion costs 13 AND triples

Comm	Setup
64-bit	13 ATs

TACEO

- Numerical approximation with Newton's method
- Sqrt includes div by secret $\rightarrow$ inv sqrt instead:

$$y_{n+1} = \frac{1}{2} y_n \left(3 - x y_n^2\right)$$

- After 3 Newton's iterations

$$\sqrt{x} = x \frac{1}{\sqrt{x}}$$

Comm	Setup
10 $\times$ 2 $\times$ 64-bit	10 MTs

Results

TACEO

- Parties set inputs
- Execute program
- Open result

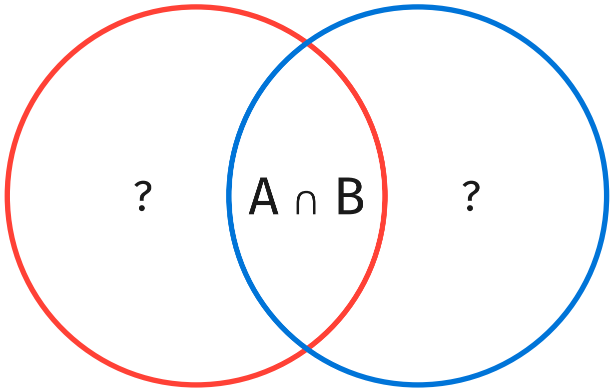
```rust
// parse RISC-V assembly
let program = "add a0, a0, a1".parse()?;
// create party with builder pattern
let mut party = PartyBuilder::new(PARTY_0, ch)
    .register_u64(XRegister::x10, Integer::Secret(2))
    .register_u64(XRegister::x11, Integer::Secret(3))
    .build()?;
// execute add function
party.execute(&program)?;
// open result in return value register
let res = party.register_u64(XRegister::x10)?;
```

- Privacy-preserving statistics
- Compute mean without revealing salaries to other party

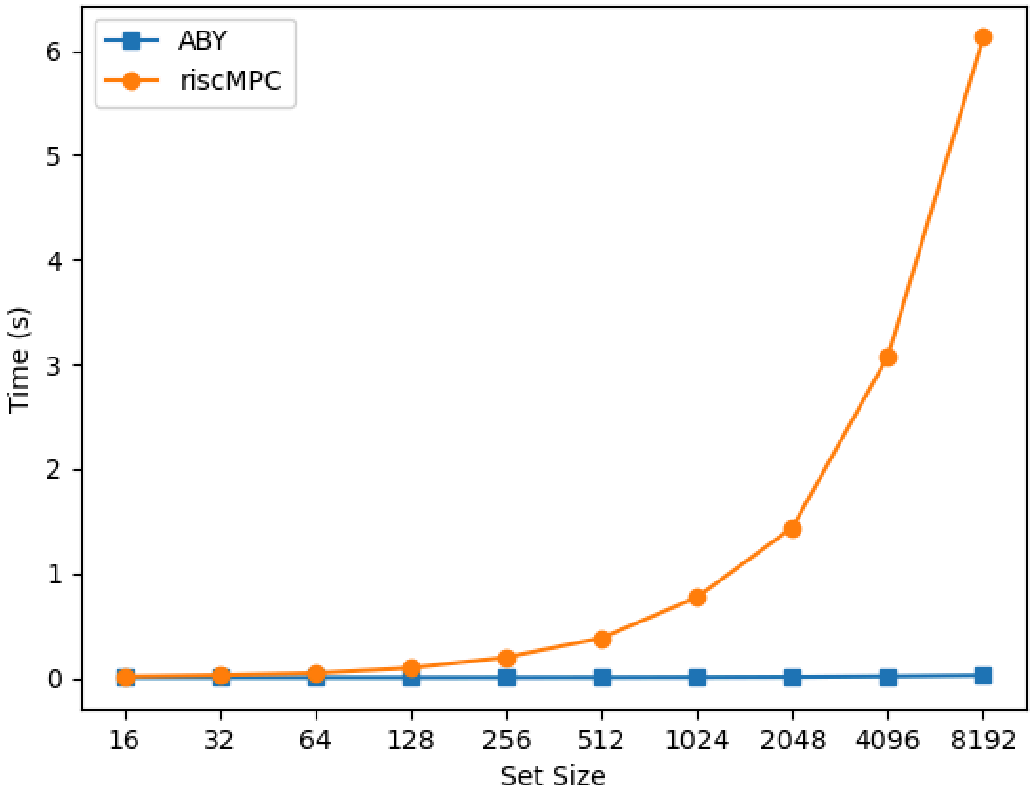
```rust
fn mean(sal0: &[u64], sal1: &[u64]) -> u64 {
  let sum =
    sal0.iter().sum::<u64>() +
    sal1.iter().sum::<u64>();
  sum / (sal0.len() + sal1.len()) as u64
}
```

Example: Private Set Intersection

- Compute intersection without revealing set to other party
- Used in contact discovery
- Different solutions exist



Example: Private Set Intersection



Example: Ascon AEAD

- Ascon round needs XOR, NOT, AND, ROT
- XOR and NOT are free
- 5 AND + 10 OR (in ROT) $\rightarrow$ 15 Beaver triples

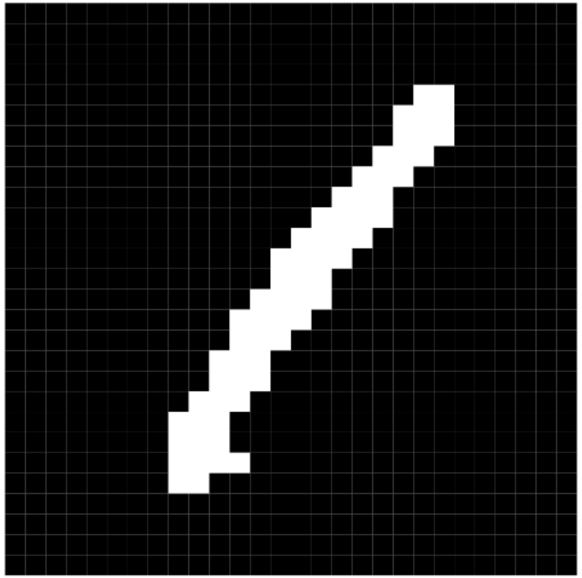
```rust
pub fn round(x: [u64; 5], c: u64) -> [u64; 5] {
    // S-box layer
    let x0 = x[0] ^ x[4];
    let x2 = x[2] ^ x[1] ^ c; // with round constant
    let x4 = x[4] ^ x[3];

    let tx0 = x0 ^ (!x[1] & x2);
    let tx1 = x[1] ^ (!x2 & x[3]);
    let tx2 = x2 ^ (!x[3] & x4);
     let tx3 = x[3] ^ (!x4 & x0);
     let tx4 = x4 ^ (!x0 & x[1]);
     let tx1 = tx1 ^ tx0;
     let tx3 = tx3 ^ tx2;
     let tx0 = tx0 ^ tx4;

     // linear layer
     let x0 = tx0 ^ tx0.rotate_right(9);
     let x1 = tx1 ^ tx1.rotate_right(22);
     let x2 = tx2 ^ tx2.rotate_right(5);
     let x3 = tx3 ^ tx3.rotate_right(7);
     let x4 = tx4 ^ tx4.rotate_right(34);
     [
         tx0 ^ x0.rotate_right(19),
         tx1 ^ x1.rotate_right(39),
         !(tx2 ^ x2.rotate_right(1)),
         tx3 ^ x3.rotate_right(10),
         tx4 ^ x4.rotate_right(7),
     ]
 }
```

TACEO

- Use pre-shared `key`
- Party 0 inputs `pt` and get `ct` + `tag`
- Party 1 doesn't learn `pt` and `ct`
- Under 1ms per block

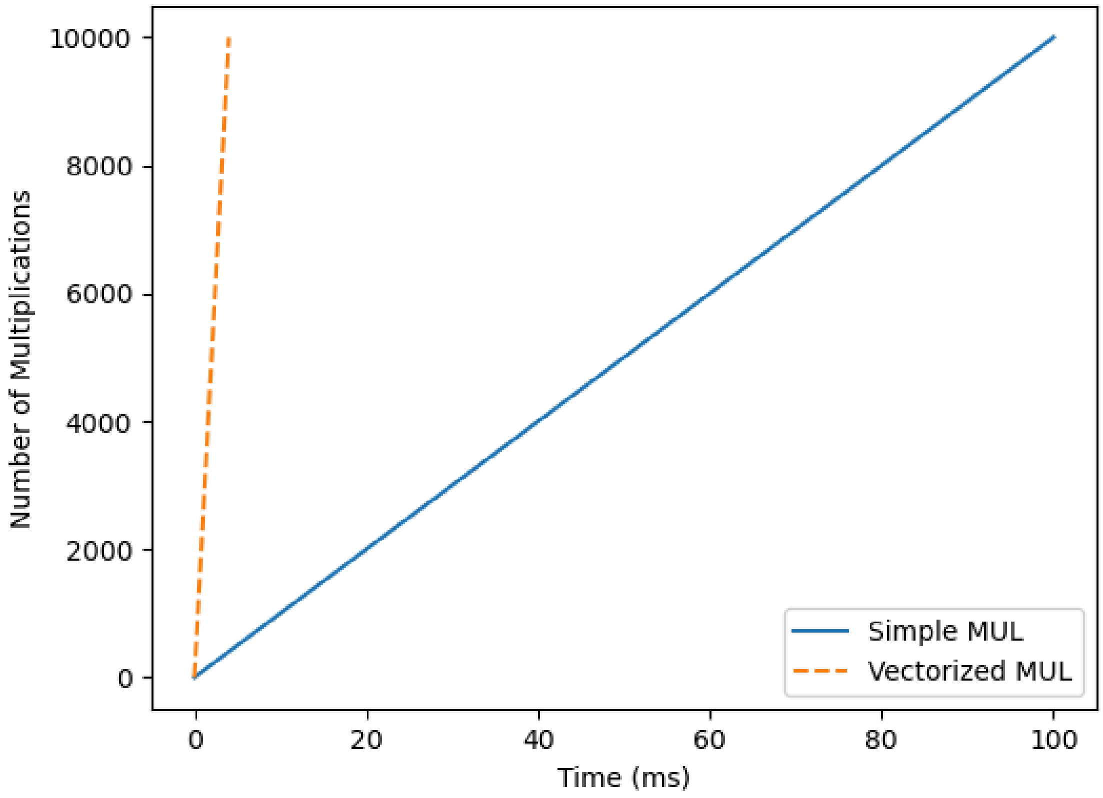
```rust
let mut party = PartyBuilder::new(PARTY_0, ch0)
  .register_u64(XRegister::x11, Integer::Public(key_addr))
  .register_u64(XRegister::x13, Integer::Public(pt_addr))
  .register_u64(XRegister::x14, Integer::Public(pt_len))
  .address_range_shared_u64(
      key_addr,
      key.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
  )?
  .address_range_u64(
       pt_addr,
       pt.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
   )?
   .n_and_triples(15 * 12 * 2 + 15 * 6 * pt_len)
   .build()?;
party.execute(
    &program.parse::<Program>()?.with_entry("encrypt_inplace")?
)?;
let ct = party.address_range_u64_for(
    pt_addr..pt_addr + pt_len * U64_BYTES, PARTY_0
)?;
let tag = party.address_range_u64_for(
    key_addr..key_addr + key_len * U64_BYTES, PARTY_0
)?;
```

- One party provides trained model, other provides image
- Inference happens in MPC
- “1” with 95.5% in 2.86s
- 100k Multiplication triples, 10k Binary triples



28x28 test image

Example: Vectorized Multiplication



Conclusion

Conclusion

- Fast development and easy to use
- RV64IM compatible
  - Support for 64-bit floating-point numbers
  - Limited support for vectorized instructions
- Good performance
  - Fast online phase
  - Fast setup phase with OT extension

Questions?

[Sha79] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[DSZ15] D. Demmler, T. Schneider, and M. Zohner, “ABY-A framework for efficient mixed-protocol secure two-party computation.,” in *NDSS*, 2015.

[Bea92] D. Beaver, “Efficient multiparty protocols using circuit randomization,” in *Advances in Cryptology—CRYPTO’91: Proceedings 11*, 1992, pp. 420–432.