

riscMPC

General-Purpose Multi-Party Computation from RISC-V Assembly

Fabian Gruber

September 06, 2024

1 Introduction

2 Background

3 riscMPC

4 Results

5 Conclusion

Introduction

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

- 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)rithmetic, (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

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



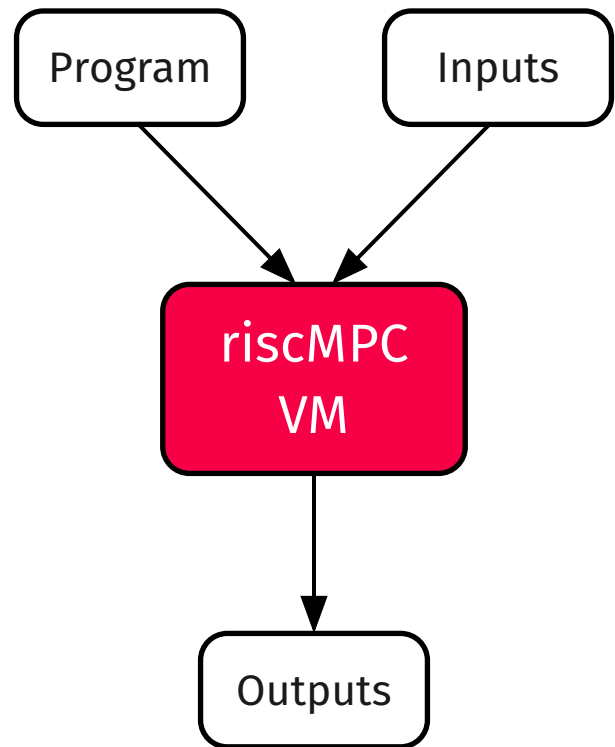
Extension	Description
“M”	Integer Multiplication and Division
“A”	Atomic Instructions
“F”	Single-Precision Floating-Point
“D”	Double-Precision Floating-Point
“Q”	Quad-Precision Floating-Point
“L”	Decimal Floating-Point
“C”	Compressed Instructions
“B”	Bit Manipulation
“J”	Dynamically Translated Languages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

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



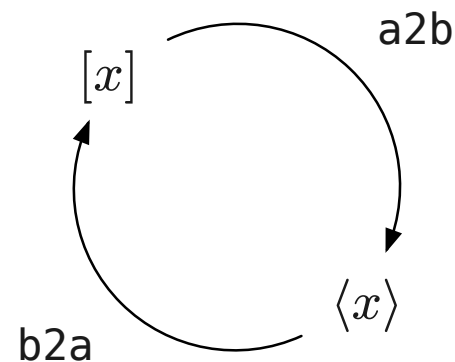
- riscMPC implements a 2-Party semi-honest MPC setting
- Both parties execute RISC-V assembly instructions
- Arguments can be secret or public



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

- Share enum represents arithmetic or binary shares

```
1 enum Share {  
2   Arithmetic(u64),  
3   Binary(u64),  
4 }
```



- Integer enum represents secret or public integers

```
1 enum Integer {  
2     Secret(Share),  
3     Public(u64),  
4 }
```

- Float enum represents secret or public floating-point numbers

```
1 enum Float {  
2     Secret(u64),  
3     Public(f64),  
4 }
```

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

- Parties locally compute $[rs1] + [rs2]$
- Public operand 1 party computes $[rs1] + rs2$

Comm	Setup
-	-

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

Comm	Setup
-	-

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

$$[d] = [\text{rs1}] - [a]$$

$$[e] = [\text{rs2}] - [b]$$

$$[\text{res}] = [c] + d[\text{rs2}] + e[\text{rs1}] + de$$

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

Comm	Setup
$2 \times 64\text{-bit}$	1 MT

- Secretly compute sign-bit:

$$[s] = [rs1] - [rs2]$$

$$\langle s \rangle = \text{A2B}([s])$$

$$\langle \text{sign-bit} \rangle = \langle s \rangle \gg 63$$

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

Comm	Setup
64-bit	13 ATs

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

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

- After 3 Newton's iterations

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

Comm	Setup
$10 \times 2 \times 64\text{-bit}$	10 MTs

Results



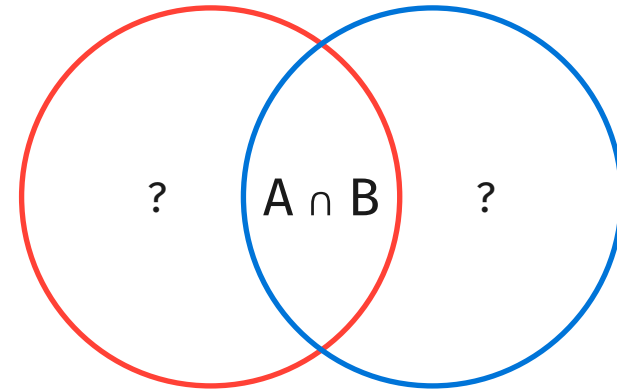
- Parties set inputs
- Execute program
- Open result

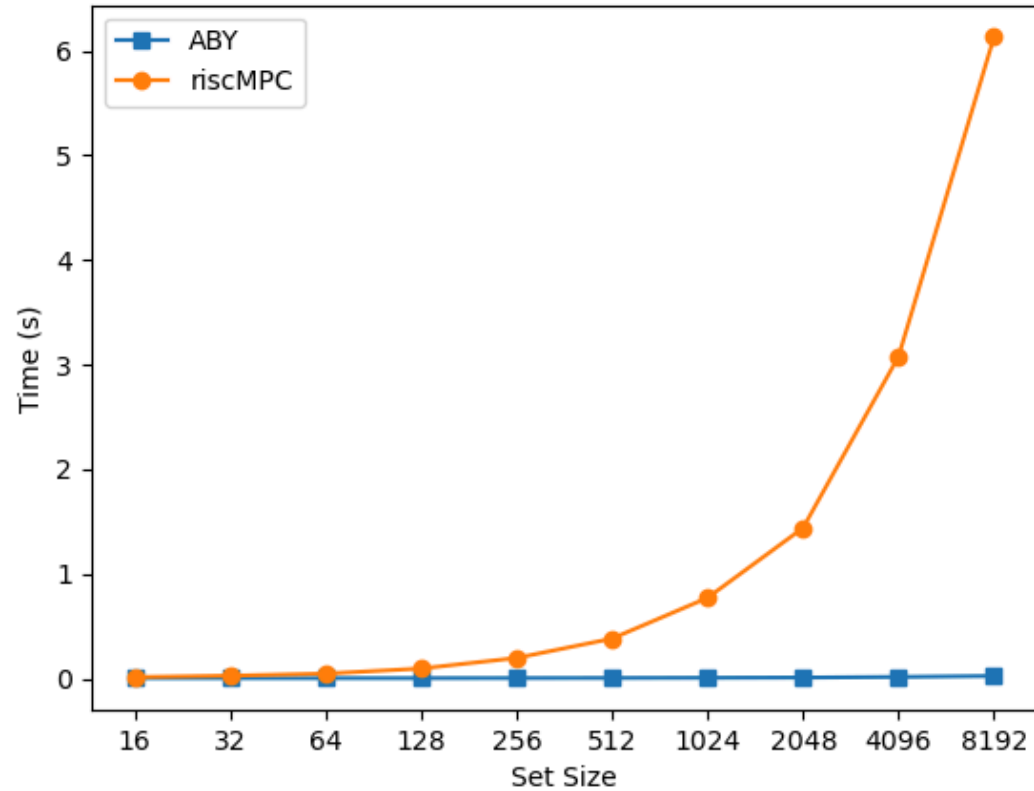
```
1 // parse RISC-V assembly
2 let program = "add a0, a0, a1".parse()?;
3 // create party with builder pattern
4 let mut party = PartyBuilder::new(PARTY_0, ch)
5     .register_u64(XRegister::x10, Integer::Secret(2))
6     .register_u64(XRegister::x11, Integer::Secret(3))
7     .build()?;
8 // execute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

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

```
1 fn mean(sal0: &[u64], sal1: &[u64]) -> u64 {  
2   let sum =  
3     sal0.iter().sum::<u64>() +  
4     sal1.iter().sum::<u64>();  
5   sum / (sal0.len() + sal1.len()) as u64  
6 }
```

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





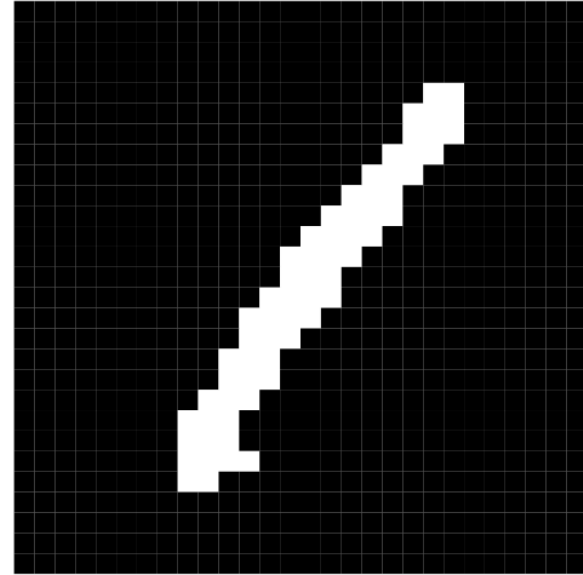
- Ascon round needs XOR, NOT, AND, ROT
- XOR and NOT are free
- 5 AND + 10 OR (in ROT) → 15 Beaver triples

```
1 pub fn round(x: [u64; 5], c: u64) -> [u64; 5] {
2     // S-box layer
3     let x0 = x[0] ^ x[4];
4     let x2 = x[2] ^ x[1] ^ c; // with round constant
5     let x4 = x[4] ^ x[3];
6
7     let tx0 = x0 ^ (!x[1] & x2);
8     let tx1 = x[1] ^ (!x2 & x[3]);
9     let tx2 = x2 ^ (!x[3] & x4);
10    let tx3 = x[3] ^ (!x4 & x0);
11    let tx4 = x4 ^ (!x0 & x[1]);
12    let tx1 = tx1 ^ tx0;
13    let tx3 = tx3 ^ tx2;
14    let tx0 = tx0 ^ tx4;
15
16    // linear layer
17    let x0 = tx0 ^ tx0.rotate_right(9);
18    let x1 = tx1 ^ tx1.rotate_right(22);
19    let x2 = tx2 ^ tx2.rotate_right(5);
20    let x3 = tx3 ^ tx3.rotate_right(7);
21    let x4 = tx4 ^ tx4.rotate_right(34);
22    [
23        tx0 ^ x0.rotate_right(19),
24        tx1 ^ x1.rotate_right(39),
25        !(tx2 ^ x2.rotate_right(1)),
26        tx3 ^ x3.rotate_right(10),
27        tx4 ^ x4.rotate_right(7),
28    ]
29 }
```

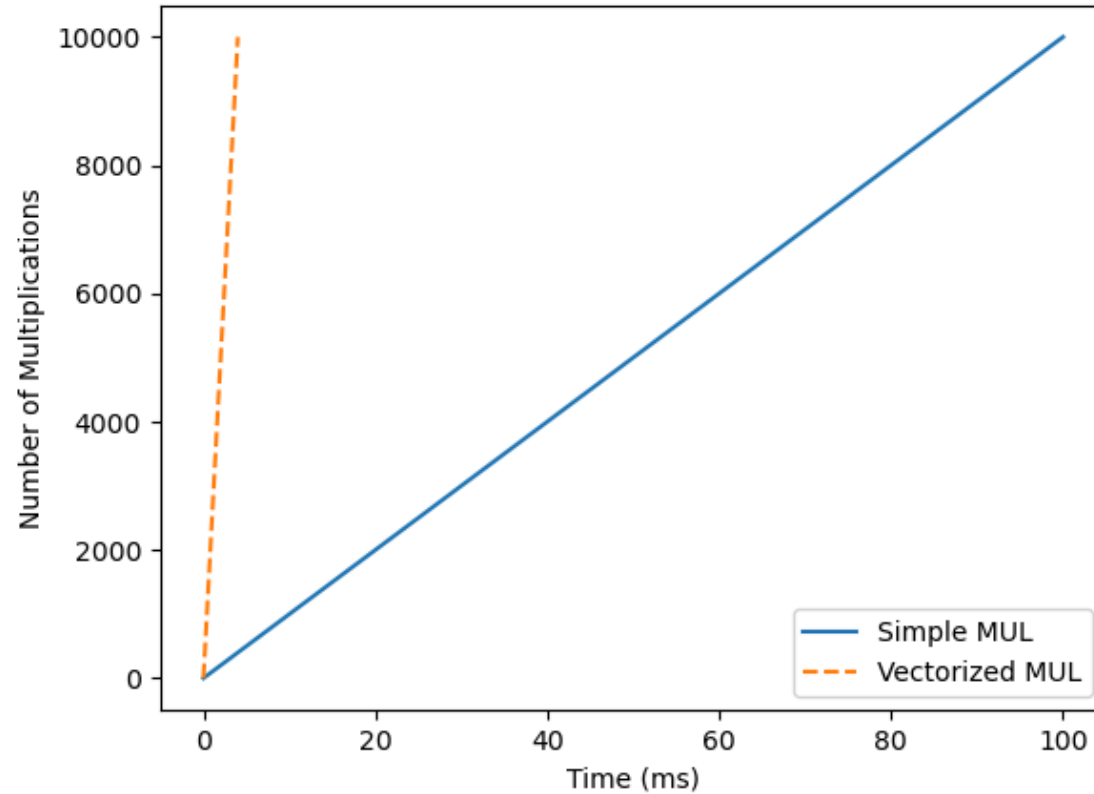
- Use pre-shared key
- Party 0 inputs pt and get ct + tag
- Party 1 doesn't learn pt and ct
- Under 1ms per block

```
1 let mut party = PartyBuilder::new(PARTY_0, ch0)
2   .register_u64(XRegister::x11, Integer::Public(key_addr))
3   .register_u64(XRegister::x13, Integer::Public(pt_addr))
4   .register_u64(XRegister::x14, Integer::Public(pt_len))
5   .address_range_shared_u64(
6     key_addr,
7     key.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
8   )?
9   .address_range_u64(
10    pt_addr,
11    pt.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
12  )?
13  .n_and_triples(15 * 12 * 2 + 15 * 6 * pt_len)
14  .build()?;
15 party.execute(
16   &program.parse:::<Program>()?).with_entry("encrypt_inplace")?
17 );
18 let ct = party.address_range_u64_for(
19   pt_addr..pt_addr + pt_len * U64_BYTES, PARTY_0
20 );
21 let tag = party.address_range_u64_for(
22   key_addr..key_addr + key_len * U64_BYTES, PARTY_0
23 );
```

- 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



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.