Cicada: Player-Scalable, Fault-Tolerant Secure MultiParty Computation

4th High-Performance Computing Security Workshop

Jon Berry, May, 2024

Project Team:

Thanks to Our Multi-Disciplinary Research Team

Jon Berry (PI)  Gabe Birch  Kevin Dixon (PM)  Anand Ganti  Ken Goss
Carolyn Mayer  Uzoma Onunkwo  Cindy Phillips  Jared Saia (UNM)  Tim Shead

Cicada-mpc
main author
Outline

• Application driver: Privacy-Preserving Machine Learning

• Algorithmic case study: dense matrix multiplication

• Software overview: Cicada-mpc (Fault-tolerant, open-source)

https://github.com/cicada-mpc/cicada-mpc/
https://cicada-mpc.readthedocs.io/
https://www.youtube.com/watch?v=GM_JuKrw4Ik
Secure MultiParty Computation

Example:

Related work for Machine Learning:

- Many others (e.g. FALCON) for 2, 3, or 4 players.

Size of circuit for ML using traditional MPC approaches (e.g. EMP) is prohibitive.
Motivation: MPC Linear Regression & Gradient Descent

Gradient descent:
Model: vector $\beta$.
Goal: Minimize a loss function $L(\beta)$ by iterating $\beta' = \beta - \eta \nabla L(\beta)$ for some learning rate $\eta$.

Why linear regression?
• Single matrix-vector multiplication in each step.
• Allows for local computations.
• Hold shares of updated model locally.
Local Gradient Matrices

Global gradient $G$ uses all datapoints.

Local gradient $G_p$ uses datapoints held by player $p$.

Then $G = \sum_p G_p$. Each $G_p$ is a share of $G$.

**Note:** Players can have different amounts of data.

**Note:** Players’ original gradient $G_p$ is additive, *but not shareable*
  - *We create “additive secret shares” that are shareable*
Typical MPC Computation: Resharing Matrices

Reshare to form matrices that don't individually reveal gradient information.

Private

Random

Received

New matrix: Private – Random + Received
For each player $p$:

1. $A_p' \leftarrow \text{AGGREGATE}(A_p, C_p)$. # sum shares along columns
2. $B_p' \leftarrow \text{AGGREGATE}(B_p, R_p)$. # sum shares along rows
3. Return $A_p' B_p'$.

Where $C_p$ is the column $p$ is in and $R_p$ is the row $p$ is in.

Coalition resisted: $\sqrt{\# \text{Players} - 1}$
**MMULT Example: 9 Players**

Aggregate $A$ in columns:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Aggregate $B$ in rows:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Local multiplications:

$\begin{align*}
\text{Global impact of MMULT:} \\
(A_1 + A_4 + A_7)(B_1 + B_2 + B_3) + (A_2 + A_5 + A_8)(B_1 + B_2 + B_3) + (A_3 + A_6 + A_9)(B_1 + B_2 + B_3) \\
(A_1 + A_4 + A_7)(B_4 + B_5 + B_6) + (A_2 + A_5 + A_8)(B_4 + B_5 + B_6) + (A_3 + A_6 + A_9)(B_4 + B_5 + B_6) \\
(A_1 + A_4 + A_7)(B_7 + B_8 + B_9) + (A_2 + A_5 + A_8)(B_7 + B_8 + B_9) + (A_3 + A_6 + A_9)(B_7 + B_8 + B_9)
\end{align*}$
Tolerating Fail-Stop Faults

Idea:

- Checkpoint row and column aggregated values.
- Use Cicada’s built-in fault tolerance and Python exception handling

```
+---+---+---+
| 1 | 2 | 4 |
+---+---+---+
| 5 | 7 | 8 |
+---+---+---+
| 9 | 11| 12|
+---+---+---+
|13|14|15|16|
```
MMULT: Theoretical Results

The Communication Complexity (CC) of MMULT is *nearly optimal* for a single matrix multiplication, and *optimal* in the amortized sense for a suite of $O(\sqrt{n})$ matrix multiplications (n is the number of players).

<table>
<thead>
<tr>
<th>Method</th>
<th>Amortized CC</th>
<th>CC</th>
<th>Coalition Res.</th>
<th>Fail-stop Tol. (GD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shamir</td>
<td>$O(nxz)$</td>
<td>$O(nxz)$</td>
<td>$k - 1 \leq n/2$</td>
<td>$n - k$</td>
</tr>
<tr>
<td>MMULT</td>
<td>$O(xy + yz)$</td>
<td>$O(\sqrt{n}(xy + yz))$</td>
<td>$\lceil \sqrt{n} \rceil - 3$</td>
<td>$\sqrt{n} - 2$</td>
</tr>
</tbody>
</table>
CICADA Software Framework

• MPC software toolkit tolerating dropouts
• Open-source:
  https://github.com/cicada-mpc/cicada-mpc/
  https://cicada-mpc.readthedocs.io/

Cooperative Computing for Autonomous DAta centers

Welcome!

Welcome to Cicada — a set of tools for working with fault-tolerant secure multiparty computation. Notable Cicada features include:

Written in Python for simplicity and ease of use.

Cicada doesn't rely on weird DSLs or runtimes, making it easier to learn, experiment, and integrate MPC computation into existing systems.

Communication inspired by the widely used MPI standard.
from cicada.communicator import SocketCommunicator

with SocketCommunicator.connect() as comm:
    print(f"Hello from player {comm.rank}!")

$ cicada run hello.py
Hello from player 0!
Hello from player 2!
Hello from player 1!
Based on three fundamental concepts

**Communicators**

Network abstraction representing an unchanging group of players, and communication patterns to pass messages among them.

**Encodings**

Map between domain values and MPC-friendly integer field representations.

**Protocol Suites**

Use communicators and encodings to implement curated collections of privacy-preserving protocols: secret sharing, addition, multiplication, logical comparison, etc.
Communication Patterns

One-to-many

Many-to-one

All-to-all

Point-to-point
Based on three fundamental concepts:

**Communicators**
Network abstraction representing an unchanging group of players, and communication patterns to pass messages among them.

**Encodings**
Map between domain values and MPC-friendly integer field representations.

**Protocol Suites**
Use communicators and encodings to implement curated collections of privacy-preserving protocols: secret sharing, addition, multiplication, logical comparison, etc.
Encoding Fixed Point Arithmetic into a Field

Use fixed number of bits and two’s complement arithmetic.

Lower order bits represent fractional part.

**Example**: 7-element field with lowest order bit representing fractional part.

Going forward, we use fixed point arithmetic in a field $F$ with a prime number of elements.
Based on three fundamental concepts:

**Communicators**

Network abstraction representing an unchanging group of players, and communication patterns to pass messages among them.

**Encodings**

Map between domain values and MPC-friendly integer field representations.

**Protocol Suites**

Use communicators and encodings to implement curated collections of privacy-preserving protocols: secret sharing, addition, multiplication, logical comparison, etc.
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

print(f"Winner: player {winner}\")
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

print(f"Winner: player {winner}"
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator

from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

    print(f"Winner: player {winner}"

Encodings
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

    print(f"Winner: player {winner}"
hostA $ cicada start --rank 0 millionaires.py
Player 0 fortune: 1230000
INFO:root:Winner: player 1

hostB $ cicada start --rank 1 millionaires.py
Player 1 fortune: 4560000
INFO:root:Winner: player 1

hostC $ cicada start --rank 2 millionaires.py
Player 2 fortune: 3400000
INFO:root:Winner: player 1
Fault Tolerance

Cicada is the only MPC library we’re aware of with support for fault tolerance and recovery!

All communication patterns have explicit, finite timeouts ...

... so failures cannot go unnoticed.

Communicators raise exceptions when failures occur ...

... this is the part where other MPC tools just die.

Applications can respond to exceptions in flexible ways ...

... communicators can be revoked (preventing subsequent use by any player)

... communicators can be shrunk (returns a new communicator with the remaining players)

... data recovery is application specific.
Thorough Documentation
Thorough Testing and Continuous Integration

7 features passed, 0 failed, 1 skipped
566 scenarios passed, 0 failed, 22 skipped
2390 steps passed, 0 failed, 111 skipped, 0 undefined
Towk 95m12.997s
MPC Through 100 Players!

![Graph showing time per full iteration (epoch) vs. number of training data points for different player counts and network settings.](image)

- **Time per full iteration (epoch)**
- **Number of training data points**

- **3 player Cicada LAN**
- **9 player Cicada LAN**
- **25 player Cicada LAN**
- **100 player Cicada LAN**
- **3 player ABY3 WAN**
- **3 player ABY3 LAN**

The graph illustrates the performance of different protocols and network conditions on the time required for one full iteration (epoch) of training as a function of the number of training data points. The data points are grouped by the number of players and network setting, showing how the time scales with the number of data points. Notably, Cicada's performance is compared with ABY, particularly in terms of timing comparisons with ABY.
Conclusions, HPC Community Asks

“WHY DIDN’T WE USE MPI and USER-LEVEL FAULT MITIGATION (ULFM)?”

Three years ago, we evaluated ULFM reference implementations in MPICH and OpenMPI. We identified problems such as:

• Communicator revocation wasn’t detected by all ranks, depending on which ranks initiated the revocation.

• Some collective operations did not raise timeout errors even when some ranks were dead.

• Because ULFM hasn’t been adopted by MPI, the Python mpi4py bindings don’t support ULFM, and working with patched bindings severely limits our ability to distribute our software.
Questions?

jberry@sandia.gov