Efficient Building Blocks For Secure Computation Based on Secret Sharing

Marina Blanton
Department of Computer Science and Engineering
University at Buffalo

Theory and Practice of Multi-Party Computation
June 17, 2019

University at Buffalo
The State University of New York

Secure Multi-Party Computation

- There are a variety of general mechanisms for securely computing on private data
- The function f being evaluated can commonly be represented as a
  - Boolean circuit
  - arithmetic circuit
    - a great number of results with various tradeoffs are available
- A fundamental question is how we build a circuit for evaluating a desired function or program f efficiently

Arithmetic Circuits

- In many instantiations the cost of addition gates is negligible compared to the cost of multiplication gates
  - thus, the number of multiplication gates is an important cost metric
  - the circuit depth is just as important to minimize
- So what is a good circuit design for simple operations such as (integer) division, shift, less-than and equality comparisons?
- With drastically different techniques such as garbled circuits, the exploration space is not as broad

Linear Secret Sharing

- Secure computation using arithmetic circuits can be realized using different techniques, but we'll talk about linear secret sharing
  - with (n, t) threshold secret sharing a secret is shared among n parties
  - access to at most t shares reveals no information about the secret
  - any linear combination of secret shared values can be carried by each share holder directly on its shares
  - multiplication is used as the basic building block
  - minimizing the number of rounds is important
    - linear round complexity for simple operations is too costly

Less-Than Comparisons

- [DFK+06] "Unconditionally Secure Constant-Rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation"
  - assumes linear secret sharing over field $\mathbb{Z}_p$ with prime p
  - provides perfect information-theoretic secrecy
  - key component: unbounded fan-in multiplication
  - comparing bit-decomposed k-bit a and b costs $22k$ invocations in 19 rounds
  - additional $100k \log_2 k + 118k$ invocations in 114 rounds are needed for each bit decomposition
  - the total is $\approx$ 40,000 invocations in 133 rounds when $k = 32$

Less-Than Comparisons

- [NO07] "Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol"
  - uses the same setting as in [DFK+06]
  - key component: open $c = a + r$ for secret a and random r
  - integrates bit decomposition with comparison
  - computed [DFK+05]'s comparison cost as $188k\log_2 k + 205k$ invocations in 44 rounds
  - developed a solution with cost $279k+5$ invocations in 15 rounds

Less-Than Comparisons

- [CdH10] "Improved Primitives for Secure Multiparty Integer Computation"
  - provides statistical instead of perfect secrecy
  - key component: opening $c = a + r$ with known bit decomposition of r
  - in addition to opening and random element generation, uses new building blocks such as generating a random integer of certain bitlength
  - achieves solutions of cost $4k - 2$ invocations in 4 rounds or $3k - 2$ invocations in 6 rounds
  - has the ability to switch between different fields for performance reasons
  - the above complexities assume non-interactive pseudo-random element generation

Catrina-de Hoogh Comparisons

- Less-than-zero comparison is specified as:

$[b] \leftarrow \mathsf{LTZ}([a], k)$

```
for $i = 0, \ldots, k-2$ do $[r_i] \leftarrow \mathsf{RandBit}(p)$;
$[r] \leftarrow \sum_{i=0}^{k-2} 2^i [r_i]$;
$[r'] \leftarrow \mathsf{RandInt}(\kappa + 1)$;
$c \leftarrow \mathsf{Open}(2^{k-1} + [a] + 2^{k-1}[r'] + [r])$;
$c' \leftarrow c \bmod 2^{k-1}$;
$[u] \leftarrow \mathsf{BitLT}(c', ([r_{k-2}], \ldots, [r_0]))$;
$[a'] = [c'] - [r] + 2^{k-1}[u]$;
$[b] \leftarrow ([a'] - [a])(2^{-(k-1)} \bmod p)$;
return $[b]$;
```

  - outputs the complement of the most significant bit of a
  - compare x and y by calling LTZ on $x - y$

Where This Takes Us

- At this point we have efficient protocols for virtually all common integer and fixed-point operations
  - different types of comparisons, truncation, division, etc.
- Can privacy-preserving evaluation of general-purpose programs be a reality?
  - we built a suite of protocols for floating-point arithmetic (NDSS'13)
    - proper evaluation of complex operations such as square root, logarithm, and exponentiation is available
  - we built a compiler, PICCO, for transforming a general-purpose C program into its secure distributed implementation (CCS'13)
    - support for dynamic memory management and pointers to private data was consequently added (TOPS'18)

Everything is Not That Simple

- The goal of the compiler was to permit programmers without extensive cryptography background create secure programs of their choice
- Experimenting with PICCO has taught us that everything is not that simple
  - knowledge of the underlying techniques is still needed for writing programs that run efficiently
- Offering built-in libraries for higher-level functions and data structures would greatly aid programmers in writing efficient code
- At a lower level, improvements can be made in two directions:
  - improving speed by using computationally secure protocols
  - exploiting computation structure to optimize more complex algorithms

Computationally Secure Protocols

- Take multiplication $a \cdot b$ as an example
  - consider (n, t) Shamir secret sharing
    - a secret s is represented by a random polynomial f of degree t with $f(0) = s$
    - each party holds evaluation of f on a unique point
  - conventional simple multiplication protocol from [GRR98] communicates $n(n-1)$ field elements local
    - locally multiply shares of a and b
    - re-share the product ($n - 1$ messages per party)
    - combine the shares and reduce the polynomial degree from $2t$ to t

Computationally Secure Multiplication

- Suppose that we use pseudorandom values for shared randomness
  - to reshare its secret, each party no longer generates a random polynomial
  - instead, each party uses PRGs to generate t shares
  - these shares together with the secret itself define the polynomial
  - now we need to communicate only $n - t - 1$ evaluations of the polynomial
  - when $n = 2t + 1$ this instantly reduces communication in half
    - with $n = 3$ and $t = 1$, this is a reduction from 6 to 3 elements per multiplication

Computationally Secure Multiplication

- We might also want to use multiplication of linear communication complexity
  - communication becomes asymmetric and uses a king
  - consider construction from [DN07]

```
$[c] \leftarrow \mathsf{Mult}([a],[b])$
1. $([r], \langle R \rangle) \leftarrow \mathsf{DRand}()$;
2. Each $p \in [1,n]$ computes $\langle D \rangle_p = [a]_p \cdot [b]_p + \langle R \rangle_p$ and sends $\langle D \rangle_p$ to the king;
3. The king reconstructs $D \leftarrow \mathsf{Open2}(\langle D \rangle)$ and sends $D$ to each party;
4. $[c] = D - [r]$;
5. return $[c]$;
```

  - this uses $2(n-1)$ messages plus the cost of DRand

Computationally Secure Multiplication

- By using computationally secure tools, we obtain non-interactive RandFld [CDI05]
  - this can generate $[r]$, but $\langle R \rangle$ needs to use independent randomness
  - randomization is possible using a fresh pseudo-random sharing of 0
    - pseudo-random $2t$-sharing $\langle 0 \rangle$ of 0 is available from [CDI05]
  - we obtain DRand implementation with no communication
    - generate $[r]$
    - multiply shares of r with shares of 1
    - rerandomize the product shares by adding $\langle 0 \rangle$
- The total multiplication communication cost with n parties is $2(n-1)$

Optimizing Algorithm’s Structure

- Take array access at private location $[j]$ as an example
  - there are two common implementations
    - multiplexer-based approach bit decomposes the index $[j]$ and selects the right element using its bits
    - comparison-based approach compares $[j]$ to each array index and chooses the one that matched
  - both have complexity $m \log m$ for an m-element array
  - PICCO implements the former, but we later determined the latter to be slightly faster

Comparison-Based Array Read

- Consider comparison-based array read at private location $[j]$

```
$[b] \leftarrow \mathsf{ArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$
1. for $i = 0$ to $m-1$ in parallel $[c_i] \leftarrow \mathsf{EQ}([j], i)$;
2. $[b] \leftarrow \sum_{i=0}^{m-1} [c_i] \cdot [a_i]$;
3. return $[b]$;
```

  - because j is compared to all indices, the computation may be redundant
  - we need to see the way equality tests are realized

Equality Testing Protocol

- Consider the following equality protocol from [ChH10]:

  $[b] \leftarrow \mathsf{EQZ}([a], k)$

  ```
  1. $([r'], [r], [r_{k-1}], \ldots, [r_0]) \leftarrow \mathsf{RandM}(k, k)$;
  2. $c \leftarrow \mathsf{Open}([a] + 2^k[r'] + [r])$;
  3. $(c_{k-1}, \ldots, c_0) \leftarrow Bits(c, k)$;
  4. for $i = 0, \ldots, k-1$ do $[d_i] \leftarrow c_i + [r_i] - 2c_i[r_i]$;
  5. $[b] \leftarrow 1 - \mathsf{KOr}([d_{k-1}], \ldots, [d_0])$;
  6. return $[b]$;
  ```

  - the cost is dominated by RandM
  - KOr has logarithmic (in k) cost

Optimizing Comparison-Based Array Access

- The first observation is that we execute EQZ on $j - i$ for fixed j and adjacent i
  - random pad is generated to protect j for $i = 0$ and open the sum as c
  - instead of generating new randomness for $i = 1$ we could simply compute it from c as $c - 1$
  - we thus open protected j and compute the values for all indices as $c - i$
  - each $c - i$ is used in consecutive computation as before
  - this reduces complexity from $O(m \log m)$ to $O(m \log \log m)$ without affecting the number of rounds

Optimizing Comparison-Based Array Access

- The resulting operation still appears to be sub-optimal
  - related values are used in a large number of KOr operations
  - the values also span most or all of possible combinations of $\log m$ bits
  - we can compute OR of all possible combinations of $\log m$ bits more efficiently than one at a time
  - our solution uses a divide-and-conquer approach:
    - divide the size into two halves, recurse on each half, then assemble the result
    - merging two sets of size 2^a and 2^b uses 2^{a+b} invocations (OR operations) in 1 round

Optimizing Comparison-Based Array Access

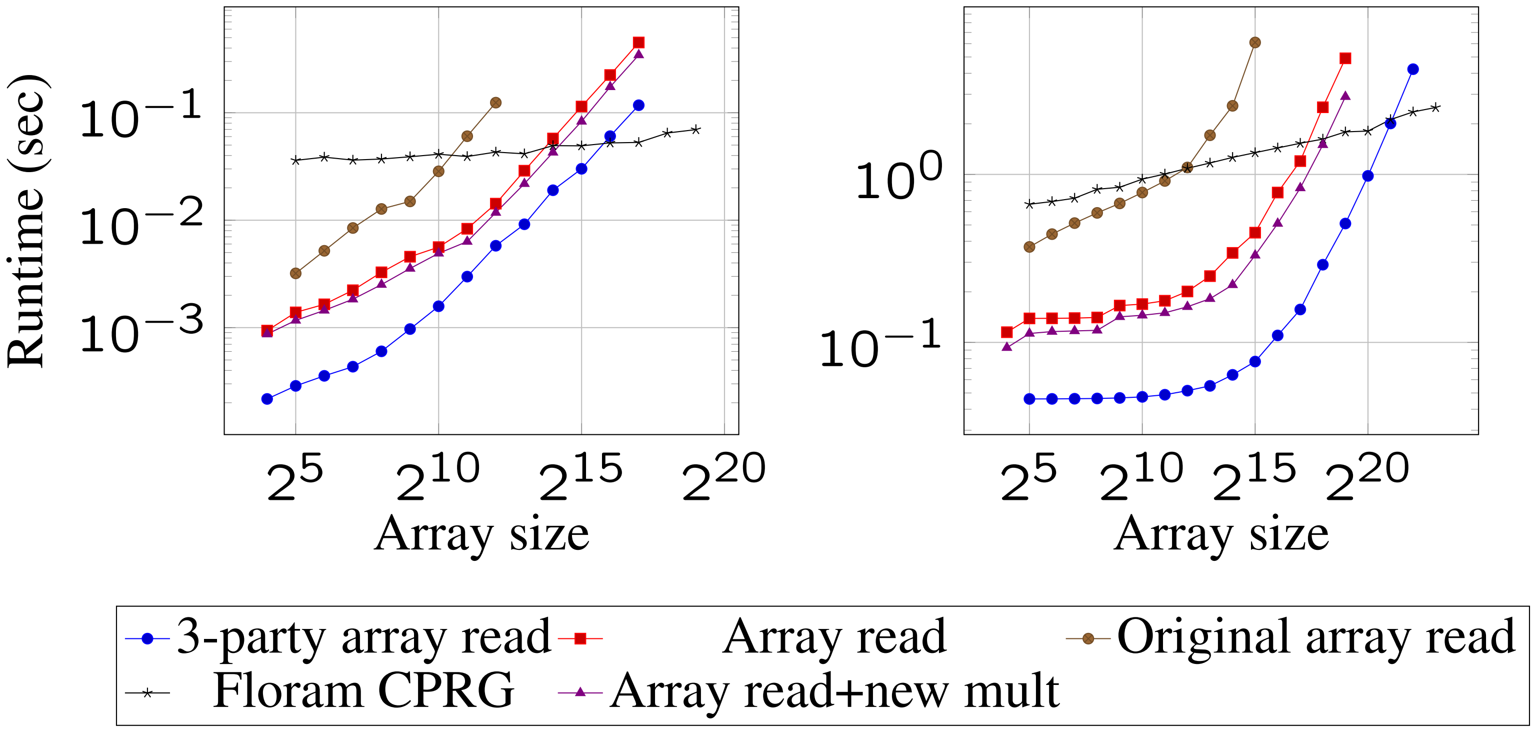
- The final solution requires some tweaks to the construction and we obtain

```
$[b] \leftarrow \mathsf{ArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$
  $([r'], [r], [r_{\log m - 1}], \ldots, [r_0]) \leftarrow \mathsf{PRandM}(\log m, \log m)$;
  $\langle [b_0], \ldots, [b_{2^{\log m} - 1}] \rangle \leftarrow \mathsf{AllOr}([r_{\log m - 1}], \ldots, [r_0])$;
  for $i = 0, \ldots, 2^{\log m} - 1$, $[b_i] = 1 - [b_i]$;
  $c \leftarrow \mathsf{Open}([j] + 2^{\log m}[r'] + [r])$;
  $c' \leftarrow c \bmod 2^{\log m}$;
  $[b] \leftarrow \sum_{i=0}^{m-1} [b_{c' - i \bmod 2^{\log m}}] \cdot [a_i]$;
  return $[b]$;
```

  - the overall complexity is $O(m)$ with a very low constant

Performance

- The impact of changes is significant in both LAN and WAN settings



Summary

- The design of common operations has a profound impact on program execution time
- Relaxing security from perfect to statistical or computational typically leads to significant performance improvements
- Working with standard secret sharing types allows for collective progress with performance of general functionalities