Securely Executing General-Purpose Programs using Secret Sharing

Marina Blanton
Department of Computer Science and Engineering
University of Notre Dame

Summer School on Secure and Oblivious Computation and Outsourcing
May 2016
Secure collaborative (multi-party) computation allows two or more parties to evaluate a function on their private inputs:

- the parties obtain their outputs, but no other information is revealed
- similar to as if the computation was performed by a trusted third party
Motivation

- **Cloud computing** enables convenient on-demand access to computing or storage resources

- **Secure computation outsourcing** allows a client to offload its task to a service provider without compromising security
  - we are interested in protecting confidentiality of client’s sensitive data used in the computation
Secure Computation Framework

- Secure computation can be realized using a number of techniques
  - secret sharing, homomorphic encryption, garbled circuit evaluation

- This lecture builds on a linear secret sharing scheme
  - provides information-theoretic security
  - uses no expensive cryptographic operations

- At high level, computation using secret sharing techniques proceeds as:
  - each private input value is split into random shares
  - secure computation proceeds on secret shares
We thus divide all participants into three groups:

- input providers/data owners
- parties that carry out the computation
- output recipients

The groups can be arbitrarily overlapping

- suitable for secure collaborative computation
- suitable for outsourcing

Input owners send shares of their data to the computational parties

Output recipients reconstruct their output from shares returned by the computational parties
Secure Computation Framework

- Using an \((n, t)\)-threshold secret sharing
  - a private value \(s\) is split into \(n\) shares
  - no information about \(s\) can be learned from any \(t\) or fewer shares
  - possession of \(t + 1\) shares allows for reconstruction of \(s\)

- With a linear secret sharing scheme
  - any linear combination of secret-shared values can be computed by each share holder locally
Secret Sharing

- **Shamir secret sharing** is of particular interest
  - it is a threshold linear secret sharing scheme
  - it can be used with any $n \geq 3$ and $t < n/2$
  - it uses a finite field $\mathbb{F}_p$ of a sufficient size

- To **generate shares** of secret integer $s$ using $(n, t)$ Shamir secret sharing
  - choose $t$ random values $a_1, \ldots, a_t$ from $\mathbb{F}_p$
  - form polynomial $f(x) = a_t x^t + \ldots + a_1 x + s$
  - compute the $i$th share as $(i, f(i))$ for $i = 1, \ldots, n$
  - distribute share $i$ to computational party $i$
• For example, let \( n = 3, t = 1, \) and \( s = 5 \) using \( \mathbb{F}_{13} \)
  
  – suppose \( a_1 = 8 \) and thus \( f(x) = 8x + 5 \)
  
  – we obtain shares \((1, 0), (2, 8), (3, 3)\)

• Given at least \( t + 1 \) shares, reconstruction of secret \( s \) uses Lagrange interpolation
  
  – suppose we have \( t + 1 \) shares \((x_0, y_0), \ldots, (x_t, y_t)\) with distinct \( x_i \)’s
  
  – to reconstruct \( s \)’s secret polynomial \( f(x) \), we compute

\[
L(x) = \sum_{i=0}^{t} y_i \ell_i(x), \quad \text{where}
\]

\[
\ell_i(x) = \prod_{0 \leq j \leq t, j \neq i} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_t}{x_i - x_t}
\]
For example, given \((1, 0)\) and \((2, 8)\), we compute \(s\)’s \(f(x)\) as:

\[
\begin{align*}
\ell_0(x) &= (x-2)(0-2)^{-1} = 11^{-1}(x-2) = 6(x-2) = 6x-12 \\
\ell_1(x) &= (x-1)(2-1)^{-1} = 1^{-1}(x-1) = x-1 \\
L(x) &= 0(6x-12) + 8(x-1) = 8x-8 = 8x+5
\end{align*}
\]

To recover only the last coefficient \(a_0 = s\) of \(f(x)\), we could compute only \(L(0)\):

\[
\begin{align*}
\text{replace } x \text{ with } 0 & \text{ in the formulas for } L(x) \text{ and } \ell_i(x) \\
\text{in our example, } L(0) &= 0 \cdot \frac{-2}{0-2} + 8 \cdot \frac{-1}{2-1} = 5 \cdot 1^{-1} = 5
\end{align*}
\]
• Any linear combination of secret shared values can be performed by each share holder locally
  – e.g., the free coefficient of the sum of two polynomials corresponds to the sum of their respective free coefficients
  – the same applies to subtraction and multiplication by a known value

• Multiplication of two secret shared values, however, is interactive
  – multiplying two shares raises the degree of the polynomial
  – the degree is brought down by all parties collectively
  – see “Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography” by Gennaro et al. for a simple multiplication procedure
  – the ability to do multiplication dictates the $t < n/2$ constraint
• **Performance** of any secret-sharing-based solution is measured in terms of two parameters
  
  – the **number of interactive operations**
    
    • these are typically multiplications
    
    • but other operations can be used as well, for example, opening a secret shared value
  
  – the **number of rounds**, i.e., sequential interactive operations
• Additions and multiplications alone are enough to support any functionality
  – it has been long known that any computable function can be represented
    as an arithmetic circuit

• Research efforts concentrated on building efficient implementations of
  common operations in this framework
  – e.g., integer comparisons are extremely important and are non-trivial to
    implement in this framework
  – the best solution for \((a < b)\) uses \(2k - 1\) interactive operations in 5
    rounds for \(k\)-bit operands
• Publications that provide the most efficient techniques for arithmetic with secret shared data:
  – “Improved Primitives for Secure Multiparty Integer Computation” by Catrina and de Hoogh, 2010
  – “Secure Computation with Fixed-Point Numbers” by Catrina and Saxena, 2010
  – “Secure Computation on Floating Point Numbers” by Aliasgari et al., 2013

• These techniques are a good starting point for enabling efficient secure evaluation of general-purpose programs
Toward General-Purpose Secure Computation and Outsourcing

- One of our projects is to enable efficient secure execution of general-purpose programs
  - general circuit-based techniques often lack efficiency
  - efficient building blocks are available for limited functionalities

- The required components are:
  - support for secure computation with standard data types
    - floating point arithmetic, string operations, etc.
  - oblivious algorithms to enable secure implementations
    - makes such operations suitable for outsourced contexts
  - a compiler that transforms a general-purpose program into its secure distributed implementation
We built a source-to-source translator called PICCO

- it transforms a user program written in extension of C to a C program that implements secure execution
- the resulting program is distributed to each computational party
- each computational party compiles secure distributed implementation by the native compiler
- input providers share their data among computational parties
- secure computation takes place on shared data
Automating General-Purpose Secure Computation

at compile time:

user
source-to-source SMC compiler
SMC config file

compute node
native compiler
executable program

at execution time:

user
runtime config file
utility program
input share
compiled program
output share
compiled program
utility program
output

config file
input
runtime config file
compiled program
compiled program
utility program

Summer School on Secure Computation
May 2016
Marina Blanton
• A user program must specify what data is to be protected
  – C is extended with the ability to mark data as private
    • e.g., `private int`, `public unsigned long`, etc.

• Standard language constructions and operations are supported, but there are restrictions due to the need to protect private data
  – loops with a private termination condition are not allowed
    • e.g., `while(priv-cond) do {...}`
  – if-statements with private conditions are not allowed to have public side effects within their scope
    • e.g., `if(priv-cond) a++`, where `a` is public
• **Round complexity** of the compiled distributed program is of paramount importance and can often be the bottleneck
  
  – existing compilers often have too restrictive or too general handling of parallelism

• PICCO has provisions for **several forms of concurrent execution** to achieve flexibility without unnecessary overhead
  
  – all elements of an array are processed simultaneously
    
    • let A, B, and C be integer arrays of the same size m
    
    • C = A * B then executes m multiplications in a single round
• Forms of concurrent execution in PICCO

  – more generally, loop iterations which are independent of each other can be processed in parallel
    • e.g., `for(i=0;i<m;i+=2)[a[i] = a[i]/a[i+1];]`
    • `[]` are used to indicate concurrent loop iterations

  – the programmer can also specify portions of the code which can be processed in parallel
    • arbitrary code can be surrounded by `[]`
    • e.g., `[a=b/c;] [d=c*c;]`
Other optimizations

- **variable-length private numeric data types** are supported
  - *e.g.*, `private int<10>, float<16,10>`
  - this improves performance of some operations
  - this also reduces the field size

- **inner product** \( \sum_{i=1}^{m} a_i \cdot b_i \) can be implemented at the cost of one interactive operation
  - this optimization often has a profound impact on performance
  - in user programs inner product is supported as \( A \oplus B \)
  - more generally, any multivariate polynomial of degree 2 can be implemented using a single interactive operation
As with any compiler, in PICCO a context-free grammar is used to parse a user’s program into an abstract syntax tree (AST)

- the AST is used to perform the necessary transformations
- declared variables and functions are stored in a symbol table

The syntax tree and the syntax table are used:

- to change all private arithmetic to be done in a field
- to compute the minimal necessary field size
- to check for data flow from private to public variables and other security violations
- to perform all necessary program transformations
To implement numeric operations on private values, the compiler links existing most efficient constructions to the transformed program
- division, bitwise operations, shifts, floating point arithmetic, etc.

The interaction of private and public values must be treated with care
- an expression that uses private data is itself private
- the compiler produces a terminal error if a private expression is being assigned to a public variable
  - a private/public status is associated with an expression in the AST
  - assigning a public value to a private variable results in converting the value to shares
Conditional statements with private conditions require special handling

- we cannot reveal anything about the private condition and thus all branches are always executed
- the effects of a branch execution may or may not be applied
- if (priv) then $a = b$ else $a = c$ is equivalent to
  $$a = \text{priv} \cdot b + (1-\text{priv}) \cdot c = \text{priv} \cdot (b-c) + c$$
- if (priv) then $a = b$ is equivalent to
  $$a = \text{priv} \cdot (b-a) + a$$
- if (priv) then ... else $a = b$ is equivalent to
  $$a = \text{priv} \cdot (a-b) + b$$
- Security requirements place restrictions on the body of conditional statements with private conditions
  - the body of such statements is not allowed to contain observable public actions
  - e.g., public variables cannot be modified
  - e.g., functions called from the body of such statements are analyzed to not have public side effects
  - a terminal error is produced if a violation is found
• Implementation of array operations
  – parallel execution of array operations requires knowledge of array size
  – transformed programs need to maintain sizes of allocated arrays

• Array access at a private index
  – this operation must be data-oblivious not to reveal any information about the index
  – in PICCO, private indexing is implemented using a multiplexer and touches all locations of the array

• e.g., for an array $a$ of size 4, the computation is

\[
(a_0 \cdot \bar{i}_0 \cdot \bar{i}_1) + (a_1 \cdot i_0 \cdot \bar{i}_1) + (a_2 \cdot \bar{i}_0 \cdot i_1) + (a_3 \cdot i_0 \cdot i_1),
\]

where $i_1 i_0$ are the bits of private index $i$
• **Parallel constructs** are implemented using a *combination of batching and threads*
  
  − parallel loop iterations are combined into batches
    
    • round complexity is the same as of running one loop iteration
  
  − if their number is larger than the threshold, several batches can be executed in separate threads
  
  − concurrent execution of arbitrary code is done via threads

• **Thread management** is realized using a *pool of threads*
  
  − each thread maintains its own queue
  
  − if a queue becomes empty, it can steal tasks from other queues
  
  − we use a dedicated thread on each machine to read all incoming data
• User input and output is handled using special I/O functions smcinput and smcoutput
  – the compiler extracts information about calls to smcinput and smcoutput in the user program
  – a utility program helps an input party to produce shares of values input into the computation
  – shares are distributed to the computational parties prior to the beginning of the computation
  – similarly, outputs are assembled using the utility program from computational parties’ shares
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Modulus size (bits)</th>
<th>PICCO LAN (ms)</th>
<th>PICCO WAN (ms)</th>
<th>Sharemind LAN (ms)</th>
<th>Sharemind WAN (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 arithmetic operations</td>
<td>33</td>
<td>0.18</td>
<td>31.6</td>
<td>71</td>
<td>203</td>
</tr>
<tr>
<td>1000 arithmetic operations</td>
<td>33</td>
<td>0.60</td>
<td>32.3</td>
<td>82</td>
<td>249</td>
</tr>
<tr>
<td>3000 arithmetic operations</td>
<td>33</td>
<td>1.60</td>
<td>34.5</td>
<td>127</td>
<td>325</td>
</tr>
<tr>
<td>5 × 5 matrix multiplication</td>
<td>33</td>
<td>0.27</td>
<td>31.6</td>
<td>132</td>
<td>264</td>
</tr>
<tr>
<td>8 × 8 matrix multiplication</td>
<td>33</td>
<td>0.45</td>
<td>32.1</td>
<td>168</td>
<td>376</td>
</tr>
<tr>
<td>20 × 20 matrix multiplication</td>
<td>33</td>
<td>2.41</td>
<td>35.7</td>
<td>1,715</td>
<td>2,961</td>
</tr>
<tr>
<td>Median, mergesort, 32 elements</td>
<td>81</td>
<td>256.7</td>
<td>6,288</td>
<td>7,115</td>
<td>22,208</td>
</tr>
<tr>
<td>Median, mergesort, 64 elements</td>
<td>81</td>
<td>649.6</td>
<td>12,080</td>
<td>15,145</td>
<td>47,636</td>
</tr>
<tr>
<td>Median mergesort, 256 elements</td>
<td>81</td>
<td>3,689</td>
<td>47,654</td>
<td>66,023</td>
<td>203,044</td>
</tr>
<tr>
<td>Median mergesort, 1024 elements</td>
<td>81</td>
<td>20,579</td>
<td>170,872</td>
<td>317,692</td>
<td>869,582</td>
</tr>
<tr>
<td>Hamming distance, 160 bits</td>
<td>9</td>
<td>0.17</td>
<td>31.1</td>
<td>72</td>
<td>188</td>
</tr>
<tr>
<td>Hamming distance, 800 bits</td>
<td>11</td>
<td>0.35</td>
<td>31.5</td>
<td>117</td>
<td>254</td>
</tr>
<tr>
<td>Hamming distance, 1600 bits</td>
<td>12</td>
<td>0.57</td>
<td>31.8</td>
<td>132</td>
<td>284</td>
</tr>
<tr>
<td>AES, 128-bit key and block</td>
<td>8</td>
<td>35.1</td>
<td>3,179</td>
<td>652</td>
<td>N/A</td>
</tr>
<tr>
<td>Edit distance, 100 elements</td>
<td>57</td>
<td>4,258</td>
<td>116,632</td>
<td>69,980</td>
<td>214,286</td>
</tr>
<tr>
<td>Edit distance, 200 elements</td>
<td>57</td>
<td>16,038</td>
<td>432,456</td>
<td>196,198</td>
<td>498,831</td>
</tr>
<tr>
<td>Fingerprint matching, 20 minutiae</td>
<td>66</td>
<td>830</td>
<td>74,704</td>
<td>24,273</td>
<td>75,820</td>
</tr>
<tr>
<td>Fingerprint matching, 40 minutiae</td>
<td>66</td>
<td>2,761</td>
<td>172,455</td>
<td>55,088</td>
<td>172,266</td>
</tr>
</tbody>
</table>
Adding Support for Pointers and Dynamic Memory

- The original PICCO design did not support all features of C
  - most notably, dynamic memory allocation and the full use of pointers were not supported

- The extended design covers all features of C
  - this brings up the notion of public and private pointers
  - this brings up handling dynamic memory allocation and deallocation
  - this also brings up the performance issues of data structures that rely on pointers
• The starting point was to determine the basic representation of pointers to private data and their secrecy type
  – performance is a key issue

• The decision was to use a single representation for all pointers to private data: location is initially public, but may become protected
  – for example, consider the code:

```c
private int *p;
p = &a;
if (priv-cond) then p = &b;
```
• Allowing for private pointer locations places constraints on user programs to maintain privacy

  – pointer to a private data cannot be assigned the address of a public variable

    ```
    p = &a;
    if (priv-cond) then p = &b;
    *p += 1;
    ```

  – in addition, pointers to public data cannot be modified inside conditional statements with private conditions
Adding Pointer Support to PICCO

- A single representation is used for all pointers to private data in transformed programs
  - the number of locations $\alpha$
  - the locations themselves $L = (\ell_1, \ldots, \ell_{\alpha})$
  - binary tags for each location $T = ([t_1], \ldots, [t_{\alpha}])$
  - additional fields such as pointer data type and the indirection level

- It is maintained that exactly one tag $t_i$ associated with the true location is set to 1 and all others are set to 0.
• **Pointer dereferencing**
  
  – we compute the value to which the pointer points as 
    \[ v = \sum_{i=1}^{\alpha} [a_i][t_i], \]
    where \([a_i]\) is the value stored at location \(\ell_i \in L\)
    
    – the dereferenced value is updated with \([a_{new}]\) by setting each
      \([a_i] = [t_i] \cdot [a_{new}] + (1 - [t_i]) \cdot [a_i]\)
  
  – the procedure for pointers to pointers is more complex

• **Pointer update**
  
  – assignments outside conditional statements with private conditions result
    in simply copying the structure
  
  – assignments inside conditional statements with private conditions
    involve merging the lists of locations and updating the tags
• **Memory allocation** for pointers to private data is performed similar to regular malloc
  
  – we define function pmalloc that additionally takes the data type
  
  – e.g., `private int* p = pmalloc(10, private int);`

• **Memory deallocation** is tricky because we don’t know what location might be in use with $\alpha > 1$
  
  – the location to be deallocated must be chosen based on public data
  
  – all other pointers that contain the location being removed must be updated to preserve correctness
Pointer-Based Data Structures

- Enabling pointer support and dynamic memory management allows for any data structure to be implemented with private data
  - linked lists, trees, stacks, queues

- Lessons learned from implementing pointer-based data structures
  - pointer-based implementations are not efficient for working with sorted data
  - with sorted data, it is better to move the data in place instead of manipulating pointers
  - in other cases, pointers result in virtually no overhead
Conclusions

- Secret sharing and other secure multi-party computation techniques allow any desired function to be evaluated securely.

- Until recently, there were no general tools for compiling general-purpose programs into equivalent secure implementations.

- PICCO is the first compiler for general-purpose programs in the context of secure multi-party computation and outsourcing.

- Unique optimizations make performance of many compiled programs particularly efficient.