SMCL

A Domain-Specific Programming Language for Secure Multiparty Computation

Janus Dam Nielsen and Michael I. Schwartzbach
Overview

• Secure Multiparty Computation

• SMCL Concepts

• An example

• Security - what, why

• Efficiency

• Future Work

• Conclusion
Secure Multiparty Computation

- $n$ parties $P_1, ..., P_n$ wish to jointly compute the computable function: $f(x_1, ..., x_n)$

- Party $P_i$ only knows the input value $x_i$ which must be kept secret from the other parties.

- Even if some adversary has power to corrupt some subset of the parties
The Millionaire's Example

Alice

SMC
Trent

Bob

June 14, 2007

Janus Dam Nielsen - PLAS 2007
SMC Solves Problems

- Auctions
- Distributed Voting
- Matchmaking
- Benchmarking
Overview

- Secure Multiparty Computation
- SMCL Concepts
- An example
- Security - what, why
- Efficiency
- Future Work
- Conclusion
Conceptual Model
Values

Clients:
- Private values
  - Booleans
  - Integers
  - Records

Server:
- Public & Secret values
  - Booleans
  - Secret booleans
  - Integers
  - Secret integers
  - Records
  - Client identity
  - Secret client identity
Communication

Clients:
- Tunnels:
  - Asynchronous
  - put and get functions
  - Primitive types only
  - Data encrypted
  - Secret data - shared and encrypted
- Functions:
  - Synchronous
  - Primitive types only
  - Invoked by server

Server:
- Tunnels:
  - Accessed via client identity
  - put and get functions
Client Identity

Clients:

Server:

Groups of clients:
- A set of clients
- All of the same kind
- Iterated using a for loop
- Uniform treatment of clients
- Secrecy of client identity
- Specified externally
Overview

- Secure Multiparty Computation
- SMCL Concepts
- An example
- Security - what, why
- Efficiency
- Future Work
- Conclusion
SMCL

The Millionaire’s Example

declare client Millionaires:
  tunnel of sint netWorth;
  
  function void main(int[] args) {
    ask();
  }
  
  function void ask() {
    netWorth.put(readInt());
  }
  
  function void tell(bool b) {
    if (b) {
      display("You are the richest!");
    } else {
      display("Make more money!");
    }
  }

declare server Max:
  group of Millionaires mills;
  
  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    
    foreach (client c in mills) {
      sint netWorth = c.netWorth.take();
      if (netWorth >= max) {
        max = netWorth;
        rich = c;
      }
    }
    
    foreach (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
  
  ask();
Overview

• Secure Multiparty Computation
• SMCL Concepts
• An example
• Security - what, why
• Efficiency
• Future Work
• Conclusion
Security

- Identity property
- Commutative property

Adversary may:
- Observe physical state of the server
- Not observe private and secret values
Adversary Traces

- A sequence of states of an entire computation
- Secret values are masked out
- Private state of clients not available
- No declassification
Identity Property

- \( p' = p'' \) - Low equiv.
- Traces must be identical
- Prevents attacks which are a function of the trace (e.g. timing)
- Requires basic operations independent of arguments
Commutative Property

- Soundness of secret representation
Ensuring Security

• Carefully crafted semantics
• Static analysis of well-typed SMCL programs
Semantics

- Conditionals are a source of differences in trace
- Execute both branches
- Termination
- Public side-effects?

```plaintext
if (b) {
    x = y;
} else {
    x = z;
}
x = b*y + (1-b)*z
```
Hoistability

- Branches must agree on public side-effects
  - Assignment to public variables
  - Communication
  - Function calls
- While loops and recursion with secret condition - not allowed
Semantic Security

- Ideal computations are inefficient
- Prove that a pragmatic version reveals same information as the ideal version
- Assist the programmer

open(e|x,y,z)
Overview

- Secure Multiparty Computation
- SMCL Concepts
- An example
- Security - what, why
- Efficiency
- Future Work
- Conclusion
### Efficiency

#### Ideal

```c
sint x = 17;
sint a = 42;
sint b = -5;
sint c = 87;
sint p = a*(x*x) + b*x +c
sint sign = 0;
int output;
if (p<0) sign = -1;
if (p>0) sign = 1;
output = open(sign|p);
```

#### Pragmatic

```c
int x = 17;
sint a = 42;
sint b = -5;
sint c = 87;
sint p = open(a*(x*x) + b*x +c|a,b,c)
sint sign = 0;
int output;
if (p<0) sign = -1;
if (p>0) sign = 1;
output = sign;
```

<table>
<thead>
<tr>
<th>(parties, threshold)</th>
<th>ideal</th>
<th>pragmatic</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,1)</td>
<td>12 sec</td>
<td>30 ms</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td>(5,2)</td>
<td>17 sec</td>
<td>65 ms</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td>(7,3)</td>
<td>30 sec</td>
<td>132 ms</td>
<td>&lt; 1 ms</td>
</tr>
</tbody>
</table>
Future Work

• Formalize Adversary traces
• Dynamic groups
• Secret compound datatypes
• More elaborate examples
Conclusion

• A DSL for SMC
• High-level abstractions
• Strong security guaranties
• Useful in practice
Questions?