Languages for Secure Multiparty Computation and Towards Strongly Typed Macros

Janus Dam Nielsen

PhD defence

May 28th, 2009

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE
AARHUS UNIVERSITY

Outline

My Work

The dissertation reports on two areas of research:

- Languages for Secure Multiparty Computation
- Towards Strongly Typed Macros

Languages for Secure Multiparty Computation

- Janus Dam Nielsen and Michael I. Schwartzbach, A Domain-specific Programming Language for Secure Multiparty Computation, Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, San Diego, California, USA, 2007
- Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigård, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Tomas Toft, and Michael I. Schwartzbach, Secure Multiparty Computation Goes Live, Proc. of Financial Cryptography, Springer-Verlag, 2009

- PySMCL, preliminary work on a variant of SMCL embedded in Python. With Ivan Damgård, Sigurd Meldgaard, Michael I. Schwartzbach

Towards Strongly Typed Macros

- Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu, Growing a Syntax, Presented at ACM SIGPLAN Foundations of Object-Oriented Languages workshop, 2009
- Towards Strongly Typed Macros, preliminary work on a strongly typed variant of the macro system presented in Growing a Syntax

Outline

Secure Multiparty Computation

- n parties $p_1, \ldots, p_n$ wish to compute a function $f(x_1, \ldots x_n)$
- Party p_i contributes input value x_i
- The computation is correct if each party get the expected output
- The computation is private if no party learns any information about the other parties input, except for what is revealed by his output of the computation.

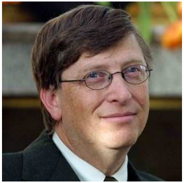
Secure Multiparty Computation

- n parties $p_1, \ldots, p_n$ wish to compute a function $f(x_1, \ldots x_n)$
- Party p_i contributes input value x_i
- The computation is correct if each party get the expected output
- The computation is private if no party learns any information about the other parties input, except for what is revealed by his output of the computation.
- And, the parties don't trust each other

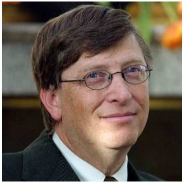
A Trusted Third Party (TTP)

1. Party p_i sends his input x_i to the TTP
2. The TTP computes the function f correctly by definition
3. The TTP sends the correct result to each of the parties and nothing more

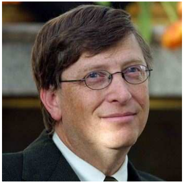
The Millionaire's Example



The Millionaire’s Example



The Millionaire’s Example



Deloitte.

The Trusted Third Party revisited

- A TTP be realized by means of cryptography

The Trusted Third Party revisited

- A TTP be realized by means of cryptography
- E.g. threshold based secret sharing scheme
  - Divide secret value x_i into shares $s^1_{x_i}, \ldots, s^n_{x_i}$
  - Distribute $s^j_{x_i}$ to party p_j
  - A threshold $1 \leq t < n$ s.t. if someone has t shares then he cannot reconstruct the secret value, but if he has $t + 1$ shares he can

Application Areas

- Auctions
- Voting
- Gambling, e.g. Poker
- Benchmarking

Outline

Why a Domain-specific Language for SMC

- Language support for fundamental concepts
- Concise description of the computation
- Fewer errors and more security
- Efficiency

A Runtime for Secure Multiparty Computation

- Share an integer in some field, e.g. $\mathbb{Z}_p$ among the parties
- Addition of two secret integers
- Multiplication of two secret integers
- Comparison of two secret integers
- Open a secret result

Conceptual Model

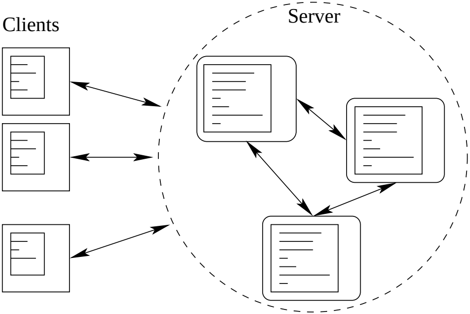


Figure: Parties are separated into clients and servers

Values

Clients have private values:

- integers
- booleans
- arrays
- records

Server has

- public values:
  - integers
  - booleans
  - arrays
  - records
  - client identity
- secret values:
  - integers
  - booleans
  - client identity

The Millionaire's Example in SMCL

```
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("Rich!");
        }
        else {
            display("Not so rich!");
        }
    }
```

```
server Max :
    group of Millionaires mills;

    function void main(int[] args) {
        sint max = 0;
        sclient rich;
        for (client c in mills) {
            sint netWorth = c.netWorth.get();
            if (netWorth > max) {
                max = netWorth;
                rich = c;
            }
        }
        for (client c in mills) {
            c.tell(open(c==rich|rich));
        }
    }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire’s Example in SMCL

```
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(“Rich!”);
        }
        else {
            display(“Not so rich!”);
        }
    }
```

```
server Max :
    group of Millionaires mills;

    function void main(int[] args) {
        sint max = 0;
        sclient rich;
        for (client c in mills) {
            sint netWorth = c.netWorth.get();
            if (netWorth > max) {
                max = netWorth;
                rich = c;
            }
        }
        for (client c in mills) {
            c.tell(open(c==rich|rich));
        }
    }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
        }
        else {
            display("Not so rich!");
        }
    }
```

```
server Max :
    group of Millionaires mills;

    function void main(int[] args) {
        sint max = 0;
        sclient rich;
        for (client c in mills) {
            sint netWorth = c.netWorth.get();
            if (netWorth > max) {
                max = netWorth;
                rich = c;
            }
        }
        for (client c in mills) {
            c.tell(open(c==rich|rich));
        }
    }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
        }
        else {
            display("Not so rich!");
        }
    }
```

```
server Max :
    group of Millionaires mills;

    function void main(int[] args) {
        sint max = 0;
        sclient rich;
        for (client c in mills) {
            sint netWorth = c.netWorth.get();
            if (netWorth > max) {
                max = netWorth;
                rich = c;
            }
        }
        for (client c in mills) {
            c.tell(open(c==rich|rich));
        }
    }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
    }
    else {
      display("Not so rich!");
    }
  }
```

```
server Max :
  group of Millionaires mills;

  function void main(int[] args) {
    sint max = 0;
    sclient rich;
    for (client c in mills) {
      sint netWorth = c.netWorth.get();
      if (netWorth > max) {
        max = netWorth;
        rich = c;
      }
    }
    for (client c in mills) {
      c.tell(open(c==rich|rich));
    }
  }
```

The Millionaire's Example in SMCL

```
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("Rich!");
        }
        else {
            display("Not so rich!");
        }
    }
```

```
server Max :
    group of Millionaires mills;

    function void main(int[] args) {
        sint max = 0;
        sclient rich;
        for (client c in mills) {
            sint netWorth = c.netWorth.get();
            if (netWorth > max) {
                max = netWorth;
                rich = c;
            }
        }
        for (client c in mills) {
            c.tell(open(c==rich|rich));
        }
    }
```

Outline

Security

- Protect input from being revealed explicitly or implicitly to any unintended entity
- The server parties work together to perform the computation
- Trust that at most a threshold t will collude against us

Information leaks

Computation may leak information

The `open` operation may leak more information, than intended

Information leaks

Computation may leak information

Examples:

- Computation time depends on secret values
- Electromagnetic radiation depends on secret values
- Interaction with hard-drive or cache depends on secret values

The `open` operation may leak more information, than intended

Information leaks

Computation may leak information

Examples:

- Computation time depends on secret values
- Electromagnetic radiation depends on secret values
- Interaction with hard-drive or cache depends on secret values

The `open` operation may leak more information, than intended

Example: $\texttt{open}(x\%10)$ and $\texttt{open}(x/10)$ reveals x

Information leaks

Computation may leak information, physical side-effects

Examples:

- Computation time depends on secret values
- Electromagnetic radiation depends on secret values
- Interaction with hard-drive or cache depends on secret values

The `open` operation may leak more information, than intended, semantic side-effects

Example: `open`$(x\%10)$ and `open`$(x/10)$ reveals x

Adversary

- May corrupt one or more server parties and clients
- Cannot break standard cryptographic assumptions
- Static but semi-honest
- Make physical measurements $m = (\phi_1, \ldots, \phi_n)$ during the execution

Physical Trace

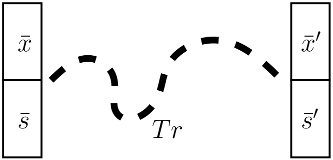
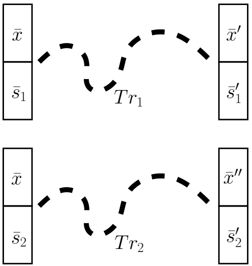


Figure: Physical trace of a computation from state $(\bar{x}, \bar{s})$ to state $(\bar{x}', \bar{s}')$

An adversary can make measurements for each step leading to a trace of measurements $Tr((\bar{x}, \bar{s})) = (m_1, \ldots, m_n)$

Identity Property

- Intuitively secure if the result of any measurements do not depend on the secret state
- The result of the measurements in $Tr((\overline{x}, \overline{s}_1))$ should be indistinguishable from result of the measurements in $Tr((\overline{x}, \overline{s}_2))$



The open Operation

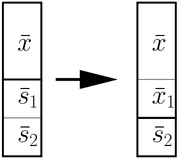


Figure: The open operation

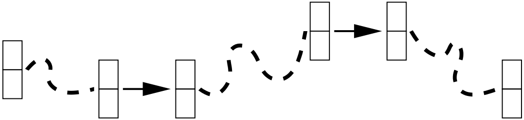


Figure: A trace sequence, an alternating sequence of physical traces and open operations

Trace Security I

Definition: An SMCL program P is said to be trace-secure if identity property holds for all physical traces of all trace sequences $(\mathrm{Tr}_1, \ldots)$ initiating from the start configuration of P

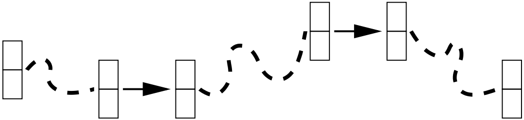


Figure: A trace sequence, an alternating sequence of physical traces and open operations

Trace Security II

Every well-typed and branch terminating SMCL Program is Trace Secure

- Execute both branches of secret-conditionals
- Branches of secret-conditional must terminate
- No assignment to public variables not declared in branches
- No I/O in branches of secret-conditionals
- No return statements in branches of secret-conditionals
- No recursion which depends on secret values
- No `for`- or `while`-loops on secret values

Towards Security Against Semantic Side-effects

Example: open$(x\%10)$ and open$(x/10)$ reveals x

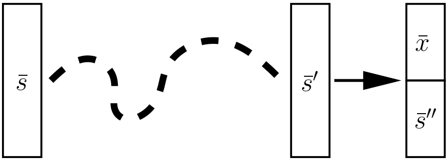


Figure: Ideal computation

- Ideal computations are secure but expensive
- Pragmatic computations may lead to unintended information leak

Towards Security Against Semantic Side-effects

Example: open$(x\%10)$ and open$(x/10)$ reveals x

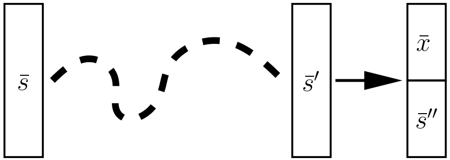


Figure: Ideal computation

- Ideal computations are secure but expensive
- Pragmatic computations may lead to unintended information leak

Checked Annotations open$(x \mid y, z)$

Future Work - PySMCL

- PySMCL a successor to SMCL embedded in Python
- Better support for security against semantic side-effects
- Let the programmer annotate the program with ideal and pragmatic `open` operations
- Generate a machine check-able proof that the values opened for pragmatic reasons can be derived from the ideally opened values.
- Use dynamic features of Python to do the embedding

Summary

- Overview of my work
- SMCL - A domain-specific language for secure multiparty computation
- SMCL programs can concisely describe secure multiparty computations using concepts unique to the secure multiparty computation domain
- SMCL programs are secure against physical side-effects
- Future work, verifiable security against semantic side-effects

Questions

Questions?