PCF: Scaling Secure Computation

Benjamin Kreuter, abhi shelat, Benjamin Mood, Kevin Butler
University of Virginia, University of Oregon
Secure 2-Party Computation

\[ f_A(x, y) \leftarrow f(x, y) \rightarrow f_B(x, y) \]

Alice

\[ x = \langle x_1 x_2 x_3 \ldots x_n \rangle \]

Bob

\[ y = \langle y_1 y_2 y_3 \ldots y_m \rangle \]

Guarantee: \( x \) and \( y \) remain private, outputs are correct

Key result [Yao82]:

Secure 2-party protocols exist for any computable function
Secure 2-Party Computation

Guarantee: $x$ and $y$ remain private, outputs are correct

Need *oblivious* representation of $f$

```plaintext
x = read_input();
if (x > 5) {
    y = 7;
} else {
    y = 12;
}
```

Leaks information about $x$
Oblivious Programs

- Control flow, memory access, etc. are independent of program inputs
- Key result: Pippenger and Fischer oblivious Turing machine construction
  - Logarithmic overhead
  - Also gives generic circuit family construction
Oblivious Programs

\[
x = \text{read\_input()};
\]
\[
\text{if} \ (x > 5) \ {\}
\]
\[
\quad y = 7;
\]
\[
\} \ \text{else} \ {\}
\]
\[
\quad y = 12;
\]
\[
\}
\]

Multiplexer

\[
x = \text{read\_input()};
\]
\[
c1 = x > 5;
\]
\[
y1 = 7;
\]
\[
c2 = !c1;
\]
\[
y2 = 12;
\]
\[
y = (y1 \ & \ c1) || (y2 \ & \ c2);
\]
Prior Work

- **Thousands** [MNPS'04]
- **Billions** [HEKM'11, KSS'12]
- **Millions** [SS'11]
- **Thousands** [MNPS'04]

Solved protocol scalability problem.

Yao'82
Prior Work

- **1982** (Yao'82)
  - Limits of Fairplay compiler

- **2004** (MNPS'04)
  - Thousands

- **2011**
  - Millions [sS'11]

- **2012**
  - Billions [HEKM'11, KSS'12]

- **2013**
  - Tens of billions
  - This work

Solves toolchain scalability problem
Solved protocol scalability problem
Tools scale poorly
Previous Approaches

- Circuits are described as lists of gates
  - Loops must be unrolled
  - Functions must be inlined
  - Conditionals must be flattened

High level domain specific lang.: SFDL/KSS/etc
MNPS04, PSSW09, KSS12, HFKV12
Problem:
Storage size = worst case running time
Previous Approaches

- Machine resources limit the size of circuits that can be optimized or stored
- Storage requirements grow with worst case running time
  - Millions of gates = many gigabytes

<table>
<thead>
<tr>
<th>Function</th>
<th>KSS12</th>
<th>HFKV12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Circuit Size</td>
<td>Compile Time</td>
</tr>
<tr>
<td>32-bit Integer Mult.</td>
<td>1.8MB</td>
<td>0.55s</td>
</tr>
<tr>
<td>1024-bit Integer Mult.</td>
<td>112MB</td>
<td>430s</td>
</tr>
<tr>
<td>16x16 Matrix Mult.</td>
<td>432MB</td>
<td>2,200s</td>
</tr>
<tr>
<td>256-bit RSA</td>
<td>15GB</td>
<td>24,000s</td>
</tr>
<tr>
<td>1024-bit RSA</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
Related problem: Evaluating Circuits

Wire values

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Related problem: Evaluating Circuits

Wire values

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Related problem: Evaluating Circuits

Wire values

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Related problem: Evaluating Circuits

Wire values

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Related problem: Evaluating Circuits

Wire values

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Related problem: Evaluating Circuits

Problem: Memory requirement grows with running time!
KSS12 Approach

- Observation: Wire values are not needed after their last use as an input to a gate

![Diagram with XOR and AND gates]

<table>
<thead>
<tr>
<th>Wire values</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
KSS12 Approach

- Observation: Wire values are not needed after their last use as an input to a gate

<table>
<thead>
<tr>
<th>Wire Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 T</td>
</tr>
<tr>
<td>T</td>
</tr>
</tbody>
</table>
KSS12 Approach

• Observation: Wire values are not needed after their last use as an input to a gate

<table>
<thead>
<tr>
<th>Wire values</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Key Insight (1): Overwriting

- When wire values are not needed, just overwrite them
- Simpler than reference counting
- The compiler can use high-level information (scope, assignments, etc.) to determine when a wire can be overwritten
- Removes need for unique wire IDs – just need the index in the table
Key Insight (2): JIT Gate Generation

- Unrolling loops carries a heavy cost: the loop body is repeated many times.
- Control-flow graphs are more compact than circuits.
  - CFGs are also useful for optimization.
Key Insight (2): JIT Gate Generation

- Loops do not need to be unrolled immediately
- Just-in-time gate generation
Key Insight (2): JIT Gate Generation

- Loops do not need to be unrolled immediately
- Gates emitted as side effects of state transitions
- Now we deal with programs, not circuits

\[
X = 0 \\
Y = 1
\]

\[
X = X + Y
\]

\[
X > Z? \\
\ldots
\]
PCF

- PCF combines these two ideas
- This requires no changes to secure 2-party computation protocols
  - The PCF runtime emits a stream of gates – this can be used like any other description
  - No compromises on security
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

loop?

...
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

Wire Table

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

loop?

...
PCF Runtime

0 := CNST(0)

1 := CNST(1)

2 := XOR(0,1)

3 := AND(0,1)

0 := COPY(2)

Wire Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

loop?

...
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

Wire Table

| 0 | 1 | 1 |

loop?

...
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

loop?

...
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

Wire Table

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

loop?

...
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

Wire Table

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

loop?

...
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

loop?

Wire Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

...
PCF Runtime

0 := CNST(0)

1 := CNST(1)

2 := XOR(0,1)

3 := AND(0,1)

0 := COPY(2)

Wire Table

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

loop?

loop?
PCF Runtime

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)

Wire Table

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

loop?

...
Key Insight (3): Start with Bytecode

- Previous systems support one language
  - Fairplay, KSS'12 use a domain specific language
  - HFKV'12 support C (suggest LLVM bytecode as future work)
- Our system reads a bytecode format as input
  - Any language can be compiled to bytecode; any language can be supported
Our System

- No changes to C, just restrictions on what programs can be executed (e.g. loop termination must not depend on input values, no pointers to functions)
- Nothing special about LCC – could use JVM, LLVM, etc. to support other languages
- Can handle big functions, tens of billions of gates or more, using just a laptop computer
unsigned int alice(unsigned int,unsigned int);
unsigned int bob(unsigned int,unsigned int);
void output_alice(unsigned int);

void main(void)
{
    unsigned int res = 0x00000001, x=0;
    unsigned int i;
    unsigned int borrow;

    for(i = 0; i < 128;)
    {
        unsigned int a1 = alice(i,0);
        unsigned int b1 = bob(i,0);
        unsigned int b = borrow;
        borrow = 0;

        if(a1 < (b1 + b))
            borrow = 1;

        i += 32;
    }

    if(borrow == 0) x = 0x1;
    else x = 0xffffffff;

    output_alice(x);
}
unsigned int alice(unsigned int, unsigned int);
unsigned int bob(unsigned int, unsigned int);
void output_alice(unsigned int);
void output_bob(unsigned int);

#define N 4
#define M 1024

unsigned int transZ[16*N];
unsigned int transO[16*N];

void read_table(void)
{
    unsigned int i = 0, inp = 0;
    for (i = 0; i < N; i++)
    {
        inp = alice(32*i, 0);
        transZ[2*i] = inp & 0xFF;
        transO[2*i] = (inp >> 8) & 0xFF;
        transZ[2*i+1] = (inp >> 16) & 0xFF;
        transO[2*i+1] = (inp >> 24) & 0xFF;
    }
}

void main(void)
{
    unsigned int i = 0, j = 0, k=0, z = 0, inp = 0;
    unsigned int state;
    read_table();
    state = 0;
    for(z = 0; z < M; z++)
    {
        inp = bob(32*z,0);
        for(i = 0; i < 32; i++)
        {
            for(j = 0; j < 16*N; j++)
            {
                unsigned int xstate = 0;
                if((inp & 0x01) != 0)
                    xstate = transO[j] & 0xFF;
                if((inp & 0x01) == 0)
                    xstate = transZ[j] & 0xFF;
                if((j == state) && (k == 0))
                {
                    k = 1;
                    state = xstate;
                }
            }
            inp = inp >> 1;
            k = 0;
        }
    output_alice(state);
}
Technical Issues
Handling Branches

- Branch statements require special treatment
  - Must evaluate all possible control paths
  - Multiplexers must be used for assignments
- In bytecode formats, branches are not as structured as in HLLs

```java
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
Handling Branches
(Fairplay/KSS'12)

```c
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```

![Diagram of branch handling]
Handling Branches (Fairplay/KSS'12)

```java
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
if(x <= 5) { 
  n = z;
  if(z == w) { 
    y += n;
  }
}
if(x <= 5) {
  n = z;
  if(z == w) {
    y += n;
  }
}
Handling (Complex) Branches

Strategy:
- Maintain global condition wire
- Emit muxes for each assignment
- Use priority queue to keep track of next branch target
Handling (Complex) Branches

GTU4

BRANCH L1

n = z;

NEU4

BRANCH L2

LABELV L1

LABELV L2

CND = true

c1 = ... > ...
Handling (Complex) Branches

CND = (!c1) && true

c1 = ... > ...

If c1 is not true, we pass this instruction

Priority Queue

(L1, c1)

LABELV L1

BRANCH L2

NEU4

n = z;

BRANCH L1

GTU4
Handling (Complex) Branches

Priority Queue

CND = (!c1) && true

mux(y, (!c1) && true)

c1 = ... > ...

(L1, c1)

LABELV L2

BRANCH L2

NEU4

BRANCH L1

GTU4

n = z;
Handling (Complex) Branches

```
GTU4
  └── BRANCH L1
      └── n = z;

NEU4
  └── BRANCH L2

LABELV L1
LABELV L2
```

```
CND = (!c1) && true

(c1 = ... > ...)

mux(y, (!c1) && true)

(c2 = ... != ...)
```
Handling (Complex) Branches

GTU4

BRANCH L1

n = z;

NEU4

BRANCH L2

LABELV L1

LABELV L2

CND = (!c2) && (!c1) && true

c1 = ... > ...

mux(y, (!c1) && true)

c2 = ... != ...

Priority order: L1 comes before L2
Handling (Complex) Branches

GTU4

BRANCH L1

n = z;

NEU4

BRANCH L2

LABELV L1

LABELV L2

CND = c1 || (!c2) && (!c1) && true

(L2, c2)

Priority Queue

C1 = ... > ...

mux(y, (!c1) && true)

c2 = ... != ...

Handling (Complex) Branches

\[
\text{CND} = c_2 \lor c_1 \lor (\neg c_2) \land (\neg c_1) ...
\]

\[
c_1 = \ldots > \ldots
\]

\[
mux(y, \neg c_1 \land \text{true})
\]

\[
c_2 = \ldots \neq \ldots
\]
Handling Loops

- Common assumption: backwards branches are used to build loops
- Only one rule: such branches must not be dependent on input values (enforced by runtime system)
- Preventing infinite loops is the user's responsibility
Circuit Optimization Strategy

- Two stages – compile time and run time
- At **compile time**, use techniques based on dataflow analysis.
  - Circuit sizes are reduced indirectly by reducing program run time
- At **run time**, check gates for constant outputs
## Circuit Optimization Strategy

<table>
<thead>
<tr>
<th>Function</th>
<th>KSS12 Total</th>
<th>KSS12 Non-XOR</th>
<th>HFKV12 Total</th>
<th>HFKV12 Non-XOR</th>
<th>This Work Total</th>
<th>This Work Non-XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>16,384-bit Millionaire's</td>
<td>98,303</td>
<td>49,154</td>
<td>330,784</td>
<td>131,103</td>
<td>97,733</td>
<td>32,229</td>
</tr>
<tr>
<td>32-bit Multiplication</td>
<td>15,935</td>
<td>5,983</td>
<td>65,121</td>
<td>26,624</td>
<td>21,742</td>
<td>6,517</td>
</tr>
<tr>
<td>64-bit Multiplication</td>
<td>64,639</td>
<td>24,384</td>
<td>321,665</td>
<td>126,529</td>
<td>105,880</td>
<td>24,766</td>
</tr>
<tr>
<td>8x8 Matrix Multiplication</td>
<td>8,067,458</td>
<td>3,058,754</td>
<td>3,267,585</td>
<td>907,776</td>
<td>1,782,656</td>
<td>522,304</td>
</tr>
<tr>
<td>16x16 Matrix Multiplication</td>
<td>64,570,969</td>
<td>24,502,530</td>
<td>24,140,673</td>
<td>7,262,208</td>
<td>14,308,864</td>
<td>4,186,368</td>
</tr>
</tbody>
</table>
## File Sizes and Compile Times

<table>
<thead>
<tr>
<th>Function</th>
<th>KSS12</th>
<th>HFKV12</th>
<th>PCF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Circuit Size</td>
<td>Compile Time (s)</td>
<td>Circuit Size</td>
</tr>
<tr>
<td>16384-bit Millionaire's</td>
<td>1.9MB</td>
<td>4.66</td>
<td>3.0MB</td>
</tr>
<tr>
<td>16000-bit Hamming Distance</td>
<td>1.9MB</td>
<td>9.75</td>
<td>9.0MB</td>
</tr>
<tr>
<td>1024-bit Multiplication</td>
<td>112MB</td>
<td>430.</td>
<td>??</td>
</tr>
<tr>
<td>16x16 Matrix Multiplication</td>
<td>432MB</td>
<td>2,200</td>
<td>206MB</td>
</tr>
<tr>
<td>256-bit RSA</td>
<td>15GB</td>
<td>24,000</td>
<td>-</td>
</tr>
<tr>
<td>1024-bit RSA</td>
<td>??</td>
<td>??</td>
<td>-</td>
</tr>
</tbody>
</table>

~1000x improvement
Comparison with “circuit libraries”

<table>
<thead>
<tr>
<th></th>
<th>Circuit Libraries [HEKM'11, MAL'11]</th>
<th>PCF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalability</strong></td>
<td>Good – Circuit not stored anywhere</td>
<td>Good – Circuit is compressed</td>
</tr>
<tr>
<td><strong>Building Circuits</strong></td>
<td>Ad-hoc – separate gadgets composed by user</td>
<td>Automatic – gadgets composed automatically by compiler</td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td>Per-gadget, user can be clever</td>
<td>Automatic, can cross gadget boundaries</td>
</tr>
</tbody>
</table>
Comparison with “circuit libraries”

- Our approach *subsumes* circuit libraries
- New gadgets can be added for new bytecode instructions
Using PCF

• A library for interpreting PCF files
• Simple interface – two functions
• Compiler and library are available upon request, and posted to github shortly
We are happy to help integrate PCF into your secure computation project
Conclusion

We have scalable protocols for secure 2-party computation

and...

We have scalable tools for secure 2-party computation
Future Work

• Other settings
  • Verifiable computation – arithmetic circuits / QAPs
  • FHE – arithmetic circuits + SIMD
  • Multiparty computation (more than 2 parties)

• Other computation models

• New optimization techniques
Questions?