Tutorial on SplitCommit and DUPLO

Roberto Trifiletti

AARHUS UNIVERSITY

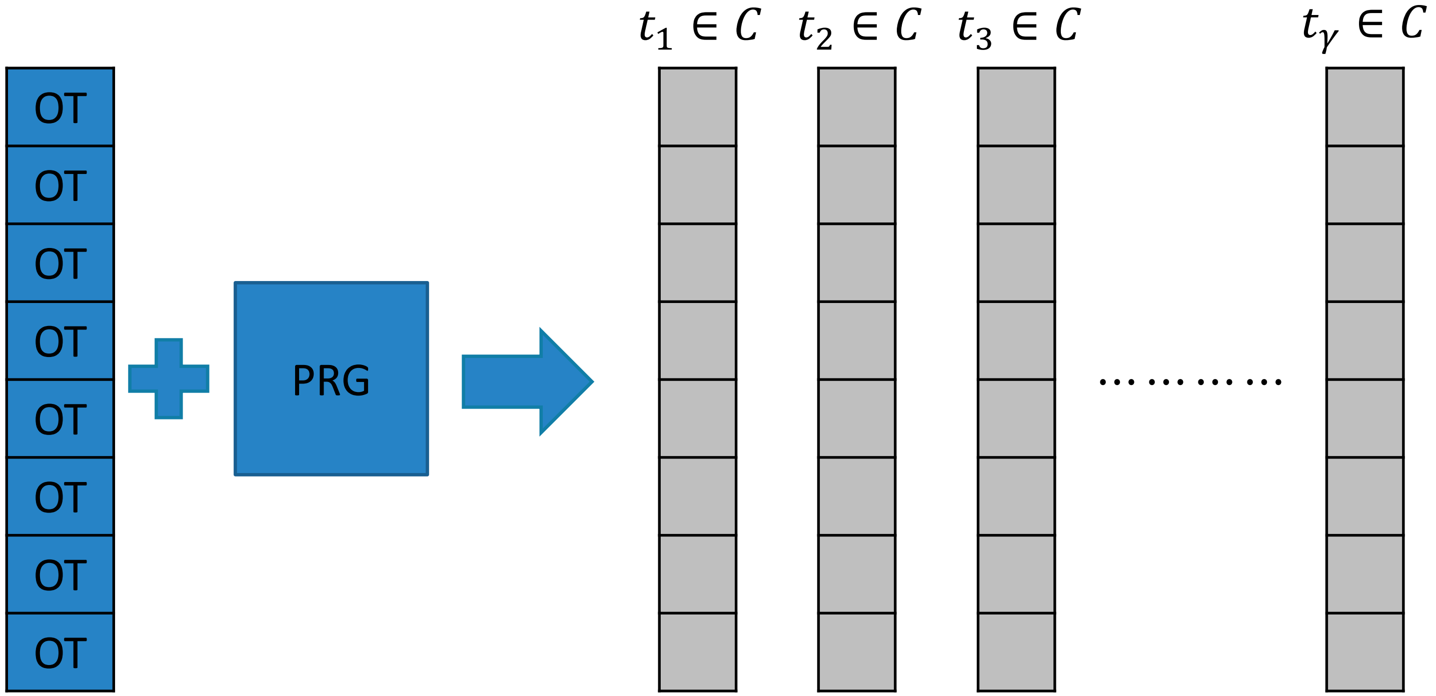
[FJNT16]

UC-secure additively homomorphic two-party commitment scheme (OT-hybrid).

- Essentially optimal rate (close to 1) for both committing and decommitting => low communication.
- After initial “seed” OTs only cheap symmetric primitives (PRG and ECC).
- Concretely efficient.

[FJNT16] Protocol in a Nutshell

Using Oblivious Transfer (OT), Linear Error Correcting Code (ECC) and a PRG the parties create a situation where



For linear ECC C with params $[n,k,d]_{\mathbb{F}}$ for $d = s$.

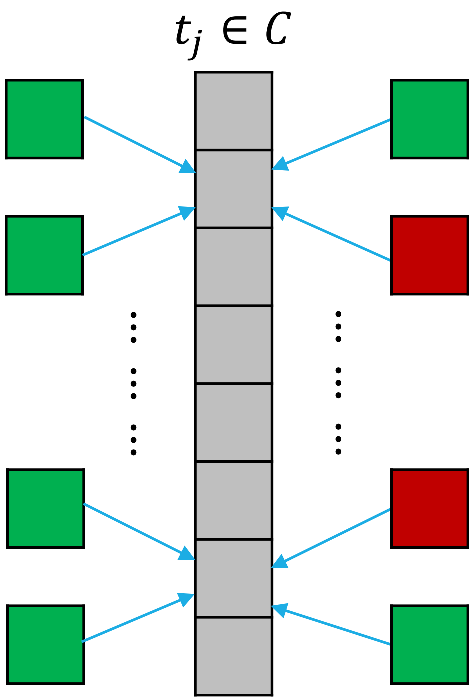
Each codeword t_j encodes a random message r_j that P_S is committing to.

[FJNT16] Protocol in a Nutshell

Furthermore for each codeword t_j:

P_S

Holds 2-additive sharing of each entry.



P_R

Holds one share for each entry of t_j. Which one is unknown to P_S, but same share-positions for all t_j.

Additively homomorphic due to linear code and additive sharing.

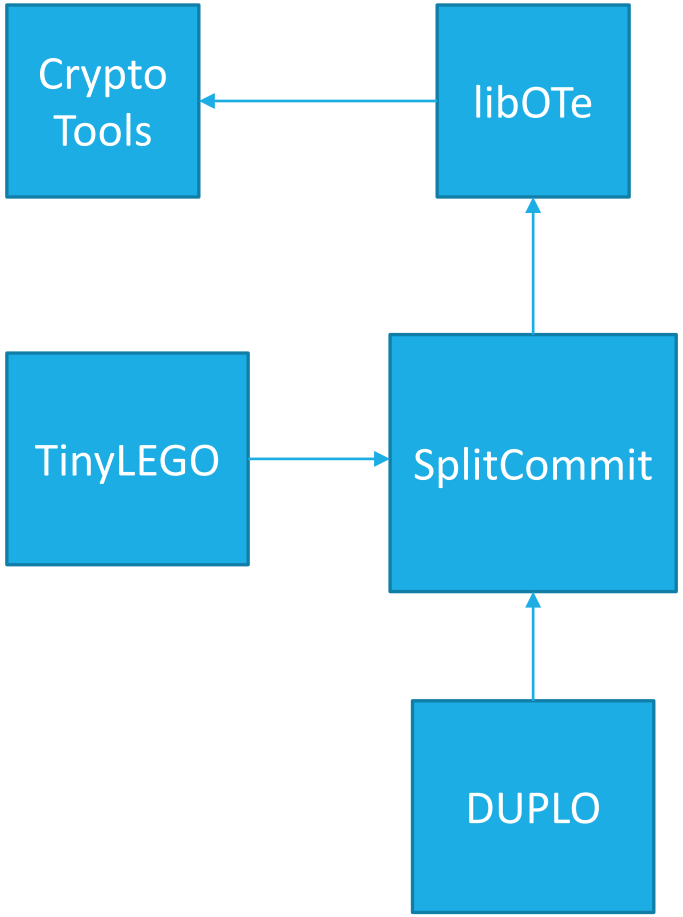
To open: P_S sends both shares of each entry and if all match and adds to codeword, P_R accepts.

Feature Highlights

For message size 16 bytes $(128 \text{ bits}), s = 40$:

- Send 17 bytes per random commitment (33 for chosen message).
- Non-interactive decommit: Send 66 bytes per decommitment.
- Allow single round of interaction: 16 bytes per decommit.
- Can decommit to subsets of committed values (e.g. lsb or all-but-1 bit).
- Can support larger messages without modifying source code, just split your message in blocks of 16 bytes and commit block-wise.
- Inserting a specific linear code would improve on communication, but requires some coding.

Overview of C++ Libraries



CryptoTools (Brent Carmer, Peter Rindal)
- Networking (Boost), efficient crypto primitives (SHA, AES), utilities.

libOTe (Brent Carmer, Peter Rindal)
- Efficient OT Extension library, both semi-honest/malicious.
- 1-out-of-2, 1-out-of-N flavors and approx. k-out-of-N.
- Offers efficient $k \times m$ bit-matrix transposition for arbitrary k, m.

SplitCommit (Peter Rindal, Roberto Trifiletti)
- [FJNT16] XOR-homomorphic commitments implementation.
- 1-bit and 128-bit string commitments availible ($s = 40$).
- Efficient way of computing linear combinations of columns of matrix.

TinyLEGO ([NST17]) and DUPLO (Roberto Trifiletti)
- Implements the LEGO 2PC protocols using SplitCommit.
- DUPLO includes tool for compiling C-like source program to decomposed boolean circuit (Ni Trieu).

SplitCommit

libOTe for OTs and CryptoTools for PRG (AES-NI).

- Get them at https://github.com/osu-crypto/libOTe and https://github.com/ladnir/cryptoTools.

[262,128,40] binary linear error correcting code (variant of BCH).

- Represented as generator matrix in .txt file.
- libOTe has functionality for efficient encoding using generator matrix representation.

Requires bit-matrix transposition of 264 × 128 bit-matrix blocks and 128 × 128 bit-matrix blocks.

- libOTe only library I know supporting $m \times n$ bit-matrix transposition efficiently for arbitrary m, n (no power of 2).

Disclaimer

Prototype software

- Not designed for real world usage in mind.
- I am not a professional developer. Probably many security bugs.
- Hardcoded seeds as source of randomness.
- Not network authentication (man-in-the-middle attack possible).

Do not use with *sensitive* data!

Works well for academic benchmarking and toy experiments.

- Hopefully inspires real world implementations.

How to Setup

Available https://github.com/AarhusCrypto/SplitCommit

- `git clone --recursive https://github.com/AarhusCrypto/SplitCommit`
- `cd SplitCommit`
- `./cmake-release.sh`
- `./build/release/TestSplitCommit` (starts a sender thread and a receiver thread and runs dummy commitments and decommitments on localhost)

roberto@cs.au.dk

How to Use (Sender)

```
SplitCommitSender commit_snd;
commit_snd.SetMsgBitSize(128);

//BaseOTs
commit_snd.ComputeAndSetSeedOTs(send_rnd, send_channel); //send_rnd is PRG object, send_channel for network

uint32_t num_commits = 1000000;

std::array<BYTEArrayVector, 2> send_commit_shares = {
  BYTEArrayVector(num_commits, 33), // 8 * 33 = 264
  BYTEArrayVector(num_commits, 33)
};

commit_snd.Commit(send_commit_shares, send_channel);

commit_snd.Decommit(send_commit_shares, send_channel); // send 2 * 33 * num_commits B, 1 round
// OR
commit_snd.BatchDecommit(send_commit_shares, send_channel); // send 2 * 33 * s + 16 * num_commits B, 2 rounds
```

How to Use (Receiver)

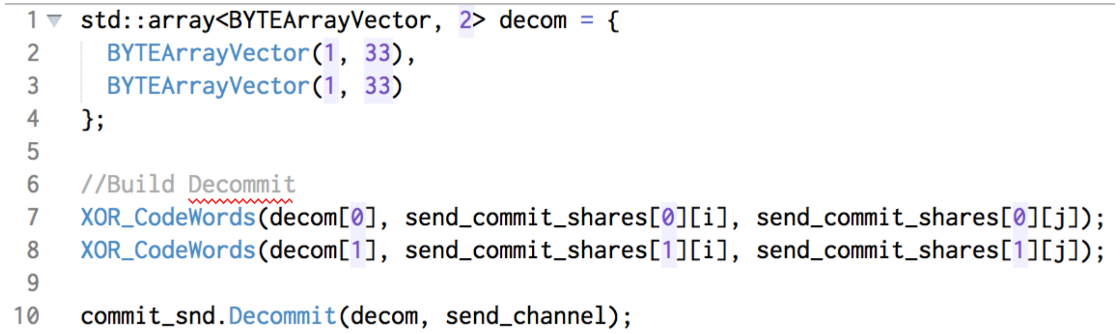
```
SplitCommitReceiver commit_rec;
commit_rec.SetMsgBitSize(128);

//BaseOTs
commit_rec.ComputeAndSetSeedOTs(rec_rnd, rec_channel); //rec_rnd is PRG object, rec_channel for network

uint32_t num_commits = 1000000;

BYTEArrayVector rec_commit_shares(num_commits, 33);

if(!commit_rec.Commit(rec_commit_shares, rec_channel)) {
  //Abort
}

BYTEArrayVector res(num_commits, 16);

if (!commit_rec.Decommit(rec_commit_shares, res, rec_channel)) {
  //Abort
}
// OR
if (!commit_rec.BatchDecommit(rec_commit_shares, res, rec_rnd, rec_channel)) {
  //Abort
}
```

XOR-homomorphism

Before decommitting, XOR the shares together and store in BYTEArrayVectors.

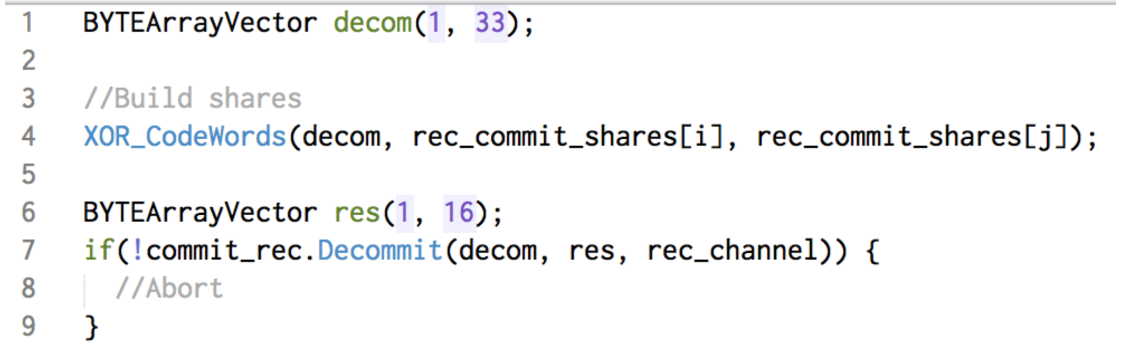
To Decommit XOR of commitments i and j:

Sender:



```
std::array<BYTEArrayVector, 2> decom = {
  BYTEArrayVector(1, 33),
  BYTEArrayVector(1, 33)
};

//Build Decommit
XOR_CodeWords(decom[0], send_commit_shares[0][i], send_commit_shares[0][j]);
XOR_CodeWords(decom[1], send_commit_shares[1][i], send_commit_shares[1][j]);

commit_snd.Decommit(decom, send_channel);
```

Receiver:



```
BYTEArrayVector decom(1, 33);

//Build shares
XOR_CodeWords(decom, rec_commit_shares[i], rec_commit_shares[j]);

BYTEArrayVector res(1, 16);
if(!commit_rec.Decommit(decom, res, rec_channel)) {
  //Abort
}
```

Performance

#Commits	Commit (incl. OTs)	Decommit	BatchDecommit
1	124,4 ms	284 us	773 us
500	249 us	0,98 us	1,38 us
1,000	125 us	1,25 us	0,86 us
15,000	9,01 us	1,02 us	0,28 us
50,000	3,26 us	1,05 us	0,33 us
500,000	0,66 us	0,92 us	0,31 us
5,000,000	0,53 us	0,8 us	0,29 us
50,000,000	0,62 us	0,78 us	0,42 us

2 x Intel Ivy Bridge i7 3.5 GHz quad-core, 32GB RAM, 1Gbit LAN.
Above for *single thread* on each machine.
Several threads: 3x faster Commit, 1,5x faster Decommit, 3x faster BatchDecommit

DUPLO

libOTe for OTs and CryptoTools for PRG (AES-NI).

SplitCommit for XOR-homomorphic commitments of 128-bit values (garbling keys).

Features:

- Frigate Extension for compiling C-like programs into decomposed boolean circuits.
- AES-NI to implement fixed-key Garbling.
- [ZRE15] HalfGates supporting any boolean gate type (not just XOR/AND).
- Includes TableGenerator for computing optimal LEGO-unit parameters (given N)
  - Currently for $s = 40$ only. Can easily be extended to arbirary s.
  - Both for SingleCut and MajorityCut buckets (also when catch-probability < 1, e.g. ${}^{1}/_{2}$ or ${}^{1}/_{4}$)
  - One-time computation, so hardcoded table in source code.

How to Setup

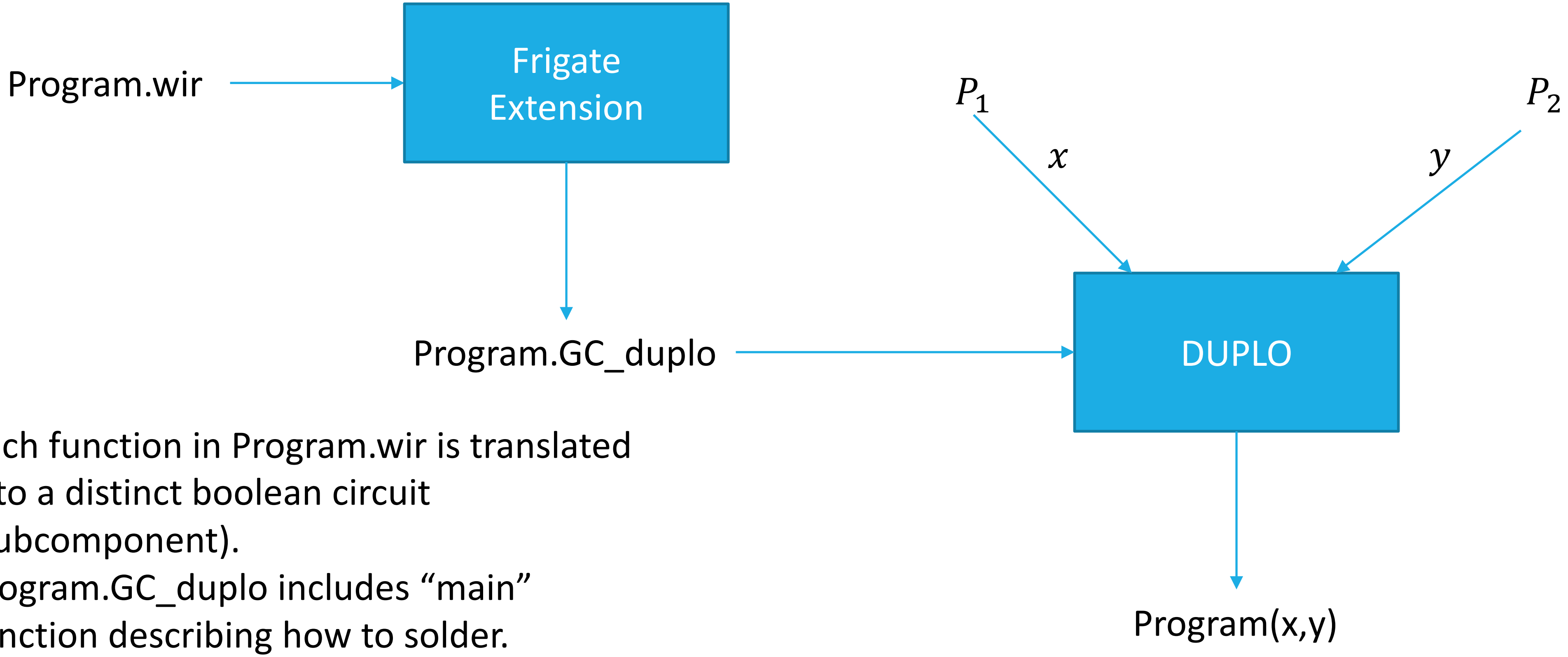
Availiable https://github.com/AarhusCrypto/DUPLO

- git clone --recursive https://github.com/AarhusCrypto/DUPLO
- cd DUPLO
- ./cmake-release.sh
- ./build/release/TestDUPLO

roberto@cs.au.dk

Provided Toolchain

We extend the recent Frigate compiler [MGCBT16] to output circuits in a format suitable for DUPLO. Same input language as Frigate (C-like syntax). Extension developed by Ni Trieu.



Program.wir

Frigate Extension

Program.GC_duplo

P_1

x

P_2

y

DUPLO

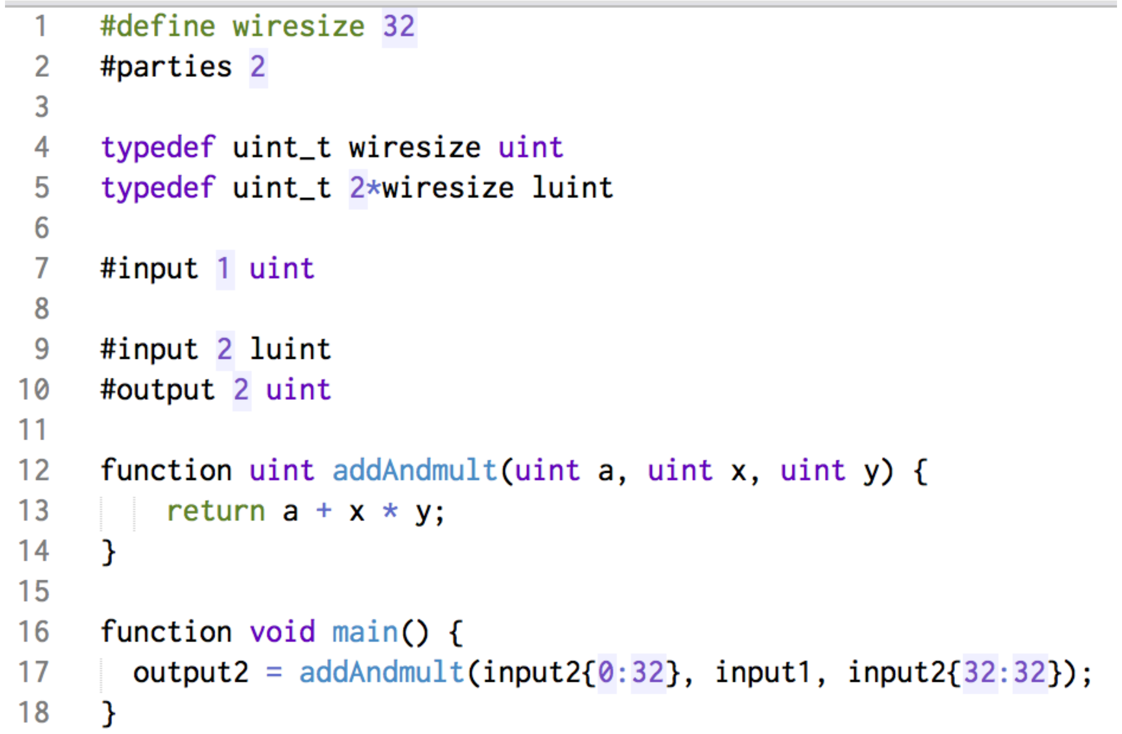
Program(x,y)

Each function in Program.wir is translated into a distinct boolean circuit (subcomponent).
Program.GC_duplo includes “main” function describing how to solder.

Program.wir (source)

Same source language as original Frigate. C dialect

- Typedefs: e.g. int is 32-bit int, lint is 64-bit int.
- Can access individual wires of variables using {idx:n} notation.
- Each function translates to distinct boolean circuit.

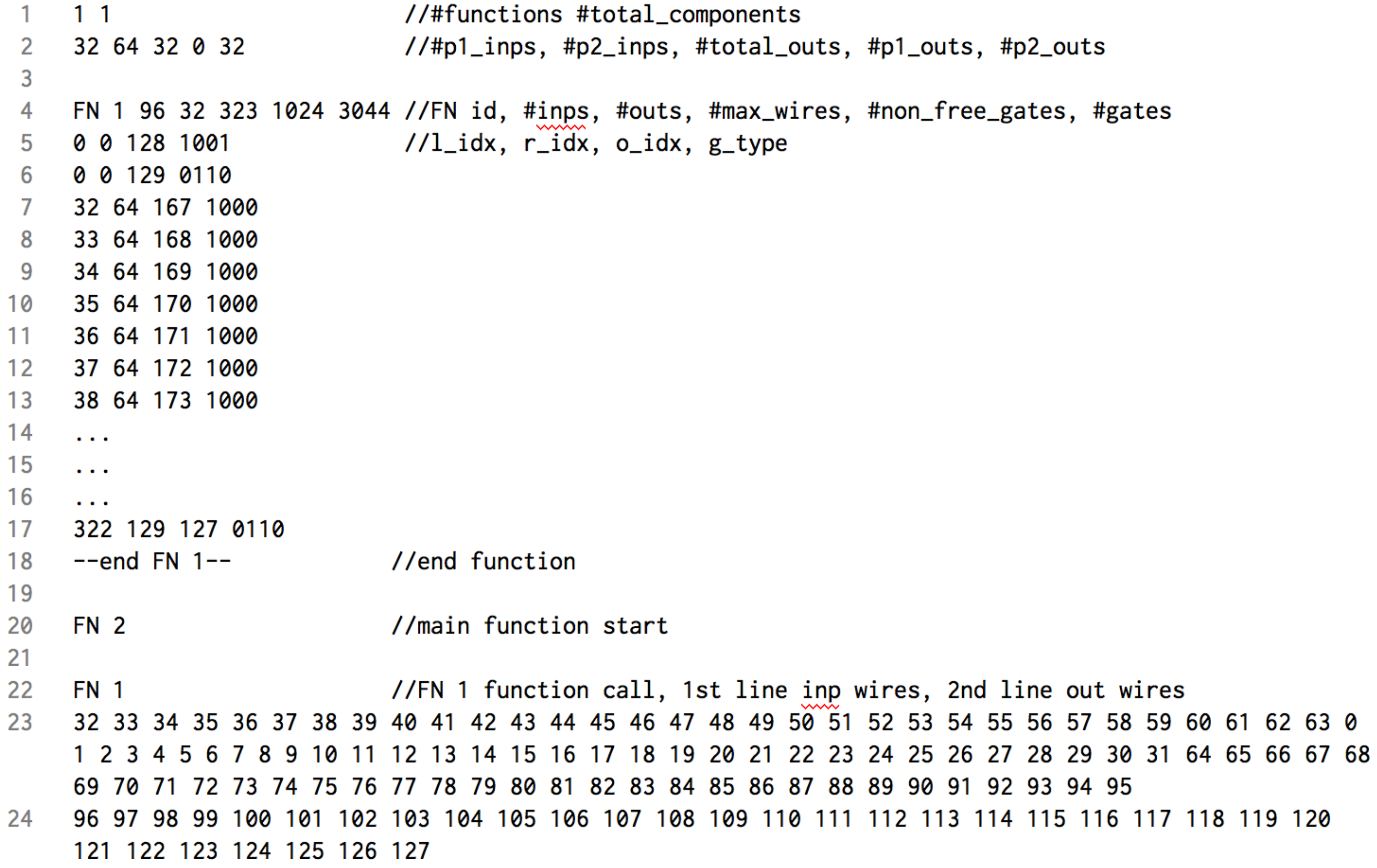


```
#define wiresize 32
#parties 2

typedef uint_t wiresize uint
typedef uint_t 2*wiresize luint

#input 1 uint

#input 2 luint
#output 2 uint

function uint addAndmult(uint a, uint x, uint y) {
    return a + x * y;
}

function void main() {
  output2 = addAndmult(input2{0:32}, input1, input2{32:32});
}
```

Program.GC_duplo (target)

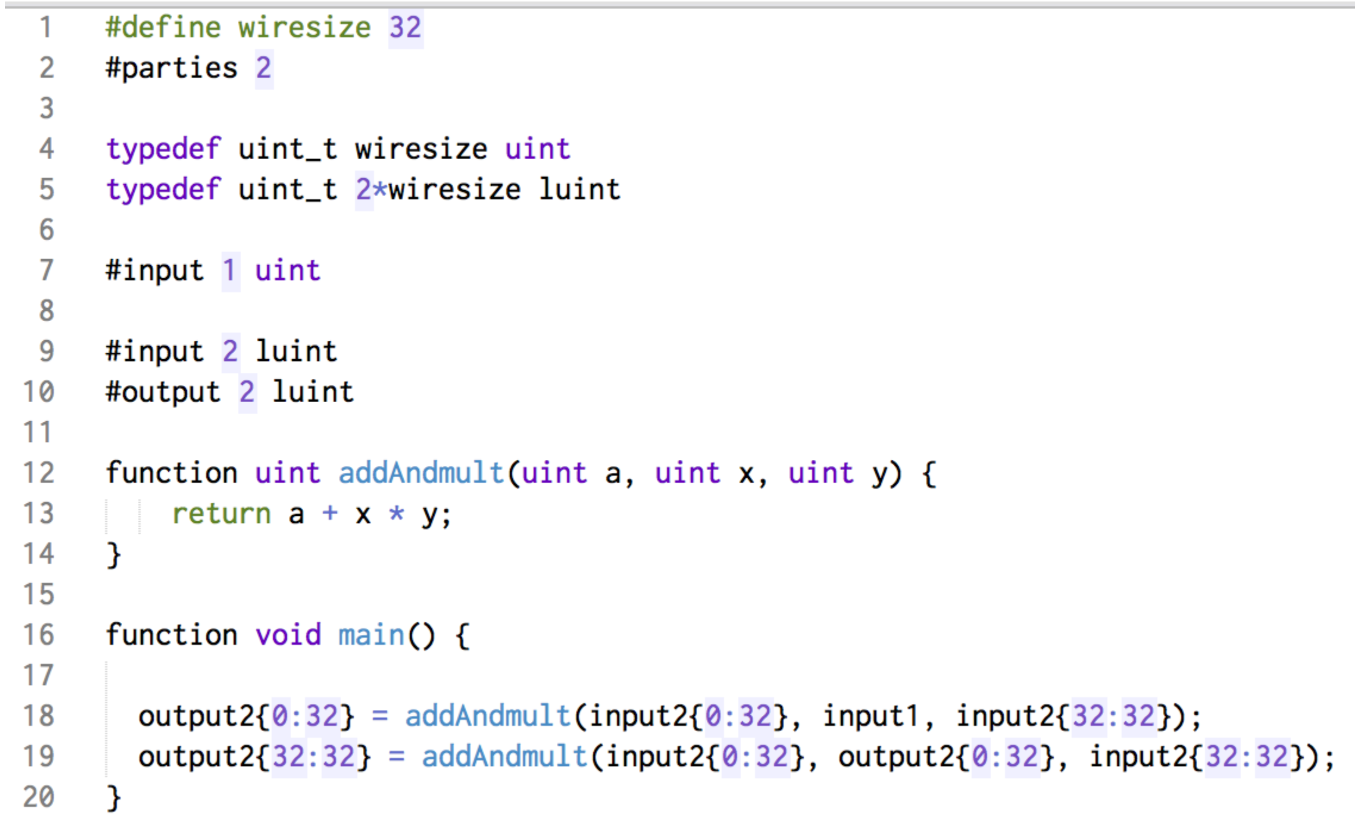
Boolean circuit file

- Enumerated circuits FN i
- Last FN is main function
- Invokes ”calls” to above defined FN circuits, keeping track of “global” wires.

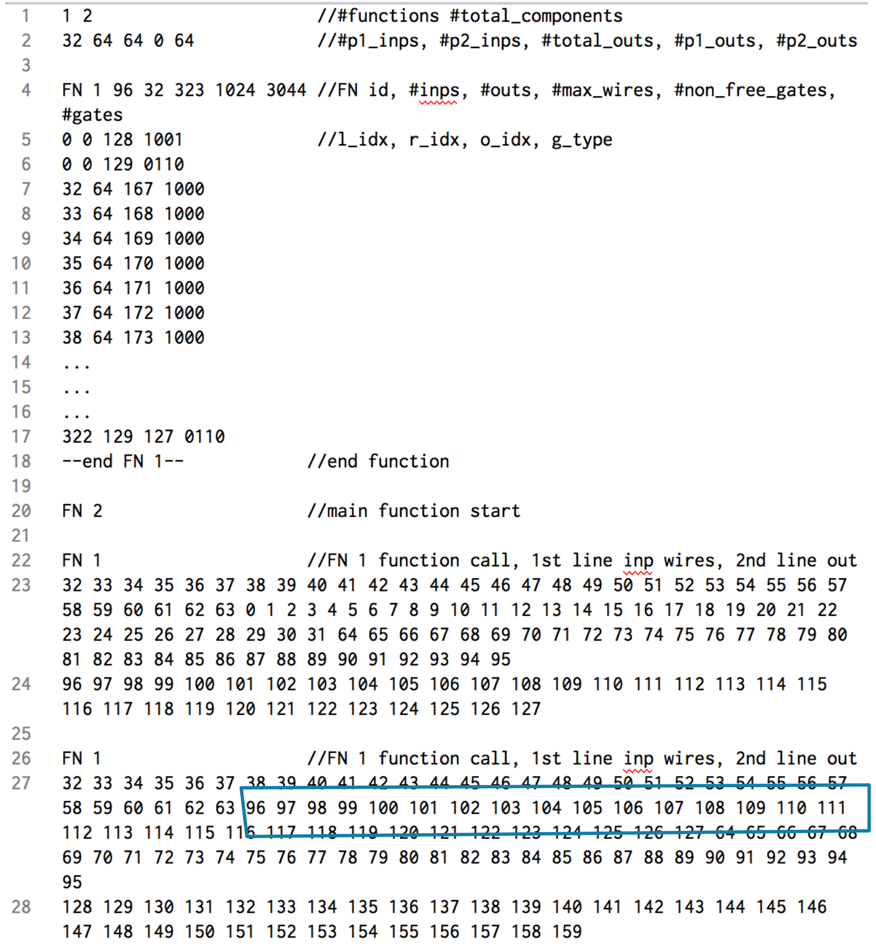


```
1 1                      //#functions #total_components
32 64 32 0 32            //#p1_inps, #p2_inps, #total_outs, #p1_outs, #p2_outs

FN 1 96 32 323 1024 3044 //FN id, #inps, #outs, #max_wires, #non_free_gates, #gates
0 0 128 1001             //l_idx, r_idx, o_idx, g_type
0 0 129 0110
32 64 167 1000
33 64 168 1000
34 64 169 1000
35 64 170 1000
36 64 171 1000
37 64 172 1000
38 64 173 1000
...
...
...
322 129 127 0110
--end FN 1--            //end function

FN 2                    //main function start

FN 1                    //FN 1 function call, 1st line inp wires, 2nd line out wires
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
```

Program.GC_duplo (target)



```
#define wiresize 32
#parties 2

typedef uint_t wiresize uint
typedef uint_t 2*wiresize luint

#input 1 uint

#input 2 luint
#output 2 luint

function uint addAndmult(uint a, uint x, uint y) {
    return a + x * y;
}

function void main() {

  output2{0:32} = addAndmult(input2{0:32}, input1, input2{32:32});
  output2{32:32} = addAndmult(input2{0:32}, output2{0:32}, input2{32:32});
}
```



```
1 2                          //#functions #total_components
32 64 64 0 64                //#p1_inps, #p2_inps, #total_outs, #p1_outs, #p2_outs

FN 1 96 32 323 1024 3044 //FN id, #inps, #outs, #max_wires, #non_free_gates,
#gates
0 0 128 1001                 //l_idx, r_idx, o_idx, g_type
0 0 129 0110
32 64 167 1000
33 64 168 1000
34 64 169 1000
35 64 170 1000
36 64 171 1000
37 64 172 1000
38 64 173 1000
...
...
...
322 129 127 0110
--end FN 1--                //end function

FN 2                        //main function start

FN 1                        //FN 1 function call, 1st line inp wires, 2nd line out
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125 126 127

FN 1                        //FN 1 function call, 1st line inp wires, 2nd line out
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
147 148 149 150 151 152 153 154 155 156 157 158 159
```

How to get Best Performance

Design program.wir in a smart way!

- Should initially optimize for LEGO-units, but not too much. It depends on concrete program, but 512-4000 LEGO-units worked well for our experiment circuits.
- Circuits should have some minimum size, the smaller the ratio (#C.inp+#C.out)/|C|, the better.
- In short, want *many* same circuits of *large* size, whenever you have 512-4000, considering increasing size.

Example: Loop unrolling

- For loop with 10,000 iterations, circuit body C with |C| = 3,000. Serially dependent.
- Construct a new loop with 2,000 iterations, each circuit body C’ does 5 iterations of C, now 2,000 LEGO-units, each of size 15,000 => similar LEGO-factor, 5x less commitments and soldering.

Thank you

Questions?

roberto@cs.au.dk

References

[FJNT16] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, Roberto Trifiletti: **On the Complexity of Additively Homomorphic UC Commitments, TCC-A 2016.**

[NST17] Jesper Buus Nielsen, Thomas Schneider, Roberto Trifiletti: **Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO, NDSS 2017.**

[ZRE15] Samee Zahur, Mike Rosulek, David Evans: **Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates, Eurocrypt 2015.**

[MGCBT16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin R. B. Butler, Patrick Traynor: **Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation, EuroS&P 2016**