Tutorial on SplitCommit and DUPLO

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

For linear ECC $C$ with params $[n, k, d]_F$ for $d = s$.

Each codeword $t_j$ encodes a random message $r_j$ that $P_S$ is committing to.
Furthermore for each codeword $t_j$:

$P_S$ holds 2-additive sharing of each entry.

$t_j \in C$

$P_R$ 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 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.
- 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 available ($s = 40$).
- 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).

[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 $264 \times 128$ bit-matrix blocks and $128 \times 128$ 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](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)

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

// Base OTs
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)

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

// Base OTs
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
} else {
    // 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:

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

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

Receiver:

```cpp
BYTEArrayVector decommit(1, 33);

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

```cpp
BYTEArrayVector res(1, 16);
if (!commit_rec.Decommit(decommit, res, rec_channel)) {
    // Abort
}
```
## Performance

<table>
<thead>
<tr>
<th>#Commits</th>
<th>Commit (incl. OTs)</th>
<th>Decommit</th>
<th>BatchDecommit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124.4 ms</td>
<td>284 us</td>
<td>773 us</td>
</tr>
<tr>
<td>500</td>
<td>249 us</td>
<td>0.98 us</td>
<td>1.38 us</td>
</tr>
<tr>
<td>1,000</td>
<td>125 us</td>
<td>1.25 us</td>
<td>0.86 us</td>
</tr>
<tr>
<td>15,000</td>
<td>9.01 us</td>
<td>1.02 us</td>
<td>0.28 us</td>
</tr>
<tr>
<td>50,000</td>
<td>3.26 us</td>
<td>1.05 us</td>
<td>0.33 us</td>
</tr>
<tr>
<td>500,000</td>
<td>0.66 us</td>
<td>0.92 us</td>
<td>0.31 us</td>
</tr>
<tr>
<td>5,000,000</td>
<td>0.53 us</td>
<td>0.8 us</td>
<td>0.29 us</td>
</tr>
<tr>
<td>50,000,000</td>
<td>0.62 us</td>
<td>0.78 us</td>
<td>0.42 us</td>
</tr>
</tbody>
</table>

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.
- Includes TableGenerator for computing optimal LEGO-unit parameters (given N)
  - Currently for $s = 40$ only. Can easily be extended to arbitrary $s$.
  - 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

Available [https://github.com/AarhusCrypto/DUPLO](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.

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.
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.
Program.GC_duplo (target)

```c
#define wiresize 32
#define parties 2

typedef uint_t wiresize uint
typedef uint_t 2*wiresize uint

#define 1 uint
#define 2 luint
#define 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
32 64 0 64
0 0 129 0 010
0 0 129 100
32 64 168 1000
34 64 168 1000
30 64 170 1000
36 64 171 1000
37 64 172 1000
38 64 173 1000
41 ...
44 ...
47 ...
50 129 127 0110
--end FN 1--
//end function

19
20
21
22
23
24
25
26
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


