SplitCommit: Implementing and Analyzing Homomorphic UC Commitments

Peter Rindal[1] and Roberto Trifiletti[2]*

[1] Oregon State University
rindalp@oregonstate.edu
[2] Aarhus University
roberto@cs.au.dk

Abstract. In this paper we present SplitCommit, a portable and efficient C++ implementation of the recent additively homomorphic commmitment scheme of Frederiksen et al. [FJNT16]. We describe numerous optimizations that go into engineering such an implementation, including highly optimized general purpose bit-matrix transposition and efficient ECC encoding given the associated generator matrix. We also survey and analyze in detail the applicability of [FJNT16] and include a detailed comparison to the canonical (non-homomorphic) commitment scheme based on a Random Oracle. We include performance benchmarks of the implementation in various network setting, for instance on a 10 Gbps LAN we achieve amortized commitment and decommitment running times of 0.65 µs and 0.27 µs, respectively. Finally we also include an extensive tutorial on how to use the library.

1 Introduction

A cryptographic commitment scheme can be thought of as the digital equivalent to a physical box with an associated lock. Imagine a setting where a sending party puts some message inside the box, locks it, and then sends it to a receiving party. As an immediate observation, since the box is locked the receiver cannot figure out what is inside by mere possession of the box. At some later point in time though, the sender can send the key that unlocks the box, and since the box has been in the receiver's custody the whole time it is guaranteed that the message inside is indeed the original message. Translating the above scenario to the digital world, we call the first property *hiding* (receiver can't look inside the box) and the second *binding* (the sender cannot change its content) and it is these properties that are realized by a cryptographic commitment scheme. We use the terms 'committing to a message' as the equivalent to sending the locked box and 'decomitting to a message' to mean sending the key that unlocks it.

In this white paper we present SplitCommit, an efficient and portable C++14 implementation of the recent additively homomorphic commitment scheme of [FJNT16]. In short, a commitment scheme is additively homomorphic if it allows the sending party to decommit to the sum of previously committed messages. This enhanced ability turns out not only to be useful in several applications, but also imply very low communication overhead when decommitting to a batch of messages.

1.1 Overview of [FJNT16]

The commitment scheme of [FJNT16] provide UC-secure additively homomorphic two-party commitments using 1-out-of-2 Oblivious Transfer (OT) as the main building block. The scheme supports committing to vectors of length k over any finite field $\mathbb{F}$, but the SplitCommit implementation put forward in this work only implements commitments for the binary field $\mathbb{F}_2 = \{0, 1\}$ and for $k \in \{1, 128\}$. In other words the source code works solely for committing to bit-strings of length 1 or 128. However, other bit-string lengths could be added in a straight forward way. Due to the binary field restriction we sometimes refer to the scheme as merely XOR-homomorphic. In addition to OT, the scheme also makes use of a linear error correction

* The first author is supported by an ARCS foundation fellowship. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA).

code (ECC) with parameters $[n, k, d]_{\mathbb{F}}$ where n is the codeword length, k is the message length, and d is the minimum distance for which we require $d \geq s$ for statistical security parameter s. As provided, the implementation only includes ECC parameters for the widely used case of $s = 40$.

The [FJNT16] scheme is tailored for committing to a batch of m messages and intuitively works in the following way. The sender picks m random messages (k-bit strings) and encode these using the above-mentioned ECC into n-bit strings. It then creates a 2-additive secret sharing of each of the resulting codewords. We view the two parts of the secret shares as making up two bit matrices, where the i'th secret sharing is located in the i'th column of the two matrices. For each m bit *row* of the two matrices, the parties perform an OT to transfer one of the two rows. As a result, for each of the n bit positions in the secret shares, the receiver learns the corresponding bit value of either the first or second secret share, and it learns the same share-position for all m codewords. Intuitively, by allowing the receiver to learn parts of each of the secret sharings, where the part that was learned is unknown to the sender, the probability for the sender to successfully change the value that it is committed to without being caught is negligible. Moreover, the receiver learns nothing about that committed value due to the receiver only learning disjoint parts of the secret sharing.

In order to later decommit to a committed value the sender sends both shares of the corresponding codeword to the receiver which can then check for each position that the value reported matches the local entry-share for this value. If they all match, and the reconstructed message is a valid codeword it accepts the message, else it aborts. In order to decommit to the XOR of messages, the sender instead sends the XOR of the corresponding shares and the receiver verifies using the XOR of its local shares. This works due to the ECC and additive secret sharing both being linear operations and therefore their composition is linear as well.

For security to hold, the scheme requires one additional step where sender gives an argument that it did not cheat. In particular a critical consistency check needs to be applied in order to ensure the sender sends secret shares that actually adds up to valid codewords. Also, in order to implement the above approach more efficiently, the parties run the 1-out-of-2 OTs on short κ-bit strings and expand these using a pseudorandom generator (PRG), as opposed to running 1-out-of-2 OTs on the potentially much longer m-bit strings which would be much less efficient.

2 Applications and Usage Guidelines

In this section we highlight some of the works in the cryptographic literature that benefit directly from an efficient XOR-homomorphic commitment scheme implementation.

The LEGO paradigm for maliciously secure two-party computation (2PC) using garbled circuits was proposed in [NO09] as an alternative to the whole-circuit Cut-and-Choose (C&C) approach of [LP07] and many following works. The central idea of whole-circuit C&C is to achieve malicious security by constructing several garbled circuits, each computing the same target function f, and checking some fraction of them for correctness. With this approach one can guarantee that when evaluating the remaining garbled circuits security and correctness holds except with probability 2^{-s} from sending $O(s)$ circuits. With LEGO, these checks are instead performed on individually garbled NAND gates, which are later combined in a secure way so that they compute f. Via a combinatorial argument it can be shown that in then suffices to send $O(\frac{s}{\log(|f|)})$ garbled circuits, a substantial asymptotic savings over whole-circuit C&C. However, in order to combine the constructed NAND gates so that they securely compute f, the gate constructor is required to commit to all wires of all garbled gates using XOR-homomorphic commitments. This is needed as it enables secure "soldering" of an output wire of one garbled gate onto the input wire of another, thus building a garbled circuit using the NAND gates as building blocks. Although the LEGO paradigm is asymptotically more efficient, the overhead of three commitments per NAND gate is substantial when looking at concrete performance and required communication. As the amount of commitments grows linearly in the size of the final circuit f it is evident that efficient constructions and implementations of XOR-homomorphic commitments are instrumental for the efficiency of any LEGO-like 2PC protocol. Recently, the work of [NST17] implemented the TinyLEGO protocol of [FJNT15] using an early version of what is now the SplitCommit library. In fact, the SplitCommit project is based on a fork of [NST17], which has since adopted it as an external library.

In addition to LEGO, XOR-homomorphic commitments have also been proposed as building blocks for evaluating RAM programs with malicious security using garbled circuits [AHMR15] and achieving non-interactive maliciously secure 2PC in the batched setting [MR17].

2.1 Usage Guidelines

It is a known fact that any UC-secure [Can01] commitment scheme in the standard model (non random oracle [BR93]) requires communicating at least the message size, in both the commitment and decommitment phase. The work of [GIKW14] was the first protocol to meet these bounds asymptotically for the non-homomorphic case, whereas [FJNT16] achieved the same for additively homomorphic commitments. It is an interesting observation, that although [FJNT16] does not achieve the above when decommitting to a single value, one can exploit the homomorphic property of the scheme to get around it when considering many or large messages. The technique is called batch opening and it works as follows for decommitting to n messages:

1. The sender sends n messages directly to the receiver, which he claims are the decommitted values.
2. The receiver then sends a challenge to the sender specifying s random linear combinations of all n messages.
3. The sender computes these s combinations, using the homomorphic property of the scheme, and decommits toward the receiver.
4. The receiver verifies the decommits and in addition computes the same combinations on the n initially claimed messages. If these match the correct decommitted combinations, he is guaranteed that the claimed messages are in fact what the sender originally committed to.

We highlight that the above technique also works for a single large message, as the sender is this case can commit to the message block-wise to obtain the same effect.

Trade-offs and Considerations. We advocate that even applications where the *homomorphic* property is not required, SplitCommit might be a right fit nonetheless. This is largely due to the concrete efficiency of the scheme as demonstrated in Section 4 and by the above batch decommit optimization. In addition, when committing to random messages, the scheme of [FJNT16] (and SplitCommit) allows for sub-linear communication in the commitment phase as these can be defined from the outputs of the PRGs directly. We here list the most profound pros and cons to consider before using the scheme:

Pros:

1. Concretely efficient, fraction of a microsecond to commit/decommit on standard hardware (see Section 4).
2. Additive homomorphism allowing for more flexibility and applicability.
3. Possibility of sub-linear communication when committing to random messages.
4. Standard-model assumptions, in particular no random oracle is required.
5. If in a context already requiring OTs (such as general 2PC), the seed OTs required for the commitment scheme can be realized using OT extension, thus greatly minimizing the initial setup cost.

Cons:

1. Two-party, directional commitments only. If the receiver needs to commit to messages as well, a separate instance of the scheme needs to be setup in the other direction.
2. Requires OT to bootstrap. Depending on setting this can be too expensive to setup, especially when only a few commitments to small messages are required.
3. Committing is interactive (single round).[3]
4. Non-interactive decommitment (non-batch) has $\sim 4\times$ communication overhead for 128-bit messages with $s = 40$.[3]

[3] For some settings, the Fiat-Shamir optimization [FS86] in the random oracle model can be applied to eliminate this interaction.

Scheme	Setup	Homomorphic	Interactive	Random Commit	Commit	Decommit	Batch Decommit
[FJNT16]	κ OTs	**Yes**	Yes	**134 bits**	262 bits	524 bits	**128 bits**
Random Oracle	**No**	No	**No**	256 bits	**256 bits**	**256 bits**	256 bits

Fig. 1. Comparison to the random oracle commitment scheme.

2.2 Comparison to Random Oracle commitments

It is widely know that a random oracle (RO) $h : \{0,1\}^* \rightarrow \{0,1\}^l$ can be used to implement a commitment scheme in a very straight-forward way. In order to commit to a message m, the sender computes $c = h(m\|r)$ for some fresh random string r and sends c to the receiver. To decommit, the sender reveals m and r to the receiver, who can then check whether $c = h(m\|r)$ or not. The above scheme can be proven secure in UC-framework if h is modeled as an extractable, programmable random oracle.[4] With this assumption the simulator in the proof of security has complete control over the hash function and can both see what values are being hashed (extract) and decide on a particular output (program). This is clearly a very strong assumption on any real-world function (and impossible to achieve in general [CGH98]), and we note for completeness that several other commitment schemes exists using weaker flavors of ROs such as [HM04,CJS14]. The latter schemes are however less efficient than the above sketched out scheme, and as such we take a worst-case approach by comparing SplitCommit to the most efficient RO based construction. Even so, it can be seen by the following discussions that the performance of SplitCommit is on par with, and sometimes supersedes that of the RO scheme on several interesting parameters.

Assumption: The RO scheme requires the existence of an extractable, programmable random oracle to be provably UC-secure, whereas SplitCommit requires the existence of UC OT. Although UC OT is also a strong starting requirement it is more generic than a random oracle.

Communication: Typically the random oracle h would be instantiated using a cryptographic hash function, say SHA-256 which then means the output size is 256 bits. Therefore no matter the size of the message (or if it's random or not), the sender needs to send 256 bits per committed value. This can be both a strength and a weakness, depending on the message size. Clearly if the message is large, the RO scheme can achieve sub-linear communication when committing, even for chosen messages. However for smaller messages, such as *e.g.* commitments to 128-bit strings, committing to a value has a 2x overhead. For decommitting, the sender needs to send the message and the random string r which is typically around 128 bits long. Again for the case of 128-bit messages this yields a 2x overhead.
For SplitCommit the cost of committing can be sublinear for random messages and essentially the message size for large chosen messages. The sublinear communication is only true for somewhat large messages and the exact cutting point depends on the security parameter and ECC parameters, but as an example, for 128-bit messages and $s = 40$ it requires 134 bits to commit to a random message and 128-bits to decommit using batch decommit.

Computation: From the performance section we see that SplitCommit can commit and decommit to a value in less than a microsecond (when considering many values). Based on the observations of [CDD$^+$15] this roughly equivalent to a SHA-256 evaluation.

Homomorphism: SplitCommit allows additively homomorphic decommits whereas the RO scheme does not.

Setup Cost: SplitCommit requires an intial setup phase involving a number of potentially expensive seed OTs. However, depending on the scope and complexity of the application this can be considered anything from a complete deal-breaker to a negligible added cost. The RO scheme requires no such setup.

Interactiveness: The RO scheme is non-interactive in both the commitment and decommitment phase, whereas SplitCommit in it's most efficient mode is interactive in both phases. However as previously mentioned, using the Fiat-Shamir heuristic (which as well requires a RO assumption) this can be turned non-interactive as well.

[4] Programmability is required to make the commitment equivocal.

3 Implementing

To obtain a high performance implementation, several of our most heavily used operations have been significantly optimized. In this section we highlight the most important ones. The most costly operation in the [FJNT16] protocol is the matrix transpose that maps m secret sharings of n bit codewords to its transpose consisting of n entries, each m bits long. In order to perform this operation efficiently, the implementation employs optimizations originally developed for OT extension protocols. The main technique is to use special instructions available on modern Intel CPUs to accelerate the computation. The most notable of these instructions is `_mm_movemask_epi8` which takes 16 bytes as input and returns a 16 bit number consisting of the least significant bit of each byte. By performing 8 of these operations interleaved with shifting the 16 bytes to the right allows for the efficient transpose of a 8×16 bit matrix, requiring roughly $20\times$ fewer instructions over a naive approach.

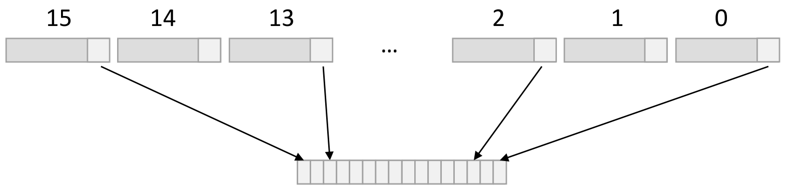

Fig. 2. Diagram of the `_mm_movemask_epi8` instruction mapping the LSB of 16 bytes to a 16 bit word.

This optimization was first observed in the cryptography community by the authors of the OT extension protocol [KOS15] and later described in a related work [KOS16]. While implementing this work and the related libOTe [Rin] project, subsequent improvements on this technique have been made. In particular, we employ Intel vector operations to allow 8 simultaneous `_mm_movemask_epi8` instructions to be performed, resulting in even greater throughput.[5] In total these optimizations yield several orders of magnitude improvements over a naive implementation and were essential in obtaining high performance.

Another computationally intensive operation is the construction of codewords that are subsequently secret shared. The most straightforward way to construct codewords $c \in \{0,1\}^n$ is by multiplying the plaintext word $p \in \{0,1\}^m$ interpreted as a bit vector ($\mathbb{F}_2^n$) by a bit matrix $G \in \{0,1\}^{m \times n}$ known as the generator. The result of this multiplication will be the desired codeword $c = Gp$. One important observation made by [FJNT16] is that a linear code can be placed in systematic form where codewords have the form $c = p||c'$, where c' is a series of parity bits that can be computed with the generator matrix G. As such, only the parity bits c' need to be computed using matrix multiplication. However, this computation can still dominate the running time if implemented naively. Namely, the naive method is to test each bit of the input and then add the corresponding row of the generator matrix to a running sum if the bit is set. When performing this operations for millions of inputs, it induces significant performance penalties due to the CPU not being able to effectively perform branch predication on the "random" input bits, thus resulting in very poor instruction pipelining.

Intuitively, we overcome this challenge by performing this conditional operation on a whole byte as opposed to single bits. This has several advantages: 1) fewer conditionals are performed. 2) branch prediction is much more effective due to being conditioned on a whole byte. 3) The resulting operations can compatible with instruction vectorization. To enable this optimization, a pre-processing on the generator matrix G is performed. G is first decomposed into $G_1, ..., G_{m/8}$ where G_i is the ith set of 8 rows of the matrix G. For each G_i and $s \in \mathbb{F}_2^8$, the value $G_i s$ is computed and stored in a lookup table. Matrix multiplication by G can

[5] Most modern Intel CPUs support 8 vector lanes.

then be reduced to

$$Gp = \sum_{i \in 1,\ldots,m/8} G_i p_i$$

,where p_i is the ith set of 8 bits in p. When implemented, the right hand side can be efficiently instantiated as $m/8$ table lookups. The overhead of this approach is the initial pre-processing cost of generating the tables, consisting of 2^8 matrix multiplications. However, when performing many commitments, this overhead is insignificant. To then achieve optimal performance, the table lookup operations can be vectorized, allowing 8 indexing operations to be performed in parallel.

The implementation as is directly supports commitments to random values. These values are obtained by first performing n so called seed OTs on random and short strings. We then use these these short strings to generate the two matrices of secret shared codewords. In particular, the seed OTs are used as seeds to such a pseudo-random number generator (PRNG) which is used to expand the seeds. Our final optimization which is widely known in the cryptography community is to use the AES-NI instruction to implement such a PRNG. In all modern Intel CPUs, the AES-NI instruction set allows for fast encryption via special hardware support. Therefore, by using AES in counter-mode, we get a hardware accelerated PRNG. As with the previous optimizations, it is critical for performance that these AES operations be vectorized, meaning at least 8 blocks of output are computed in parallel.

4 Performance

To demonstrate the scalability of the implementation, we perform benchmarks on an Intel Xeon server consisting of 2x 36-core Intel Xeon CPUs (E5-2699 v3 @ 2.30GHz) and 256GB of RAM. In the single threaded setting on a 10Gbps LAN, $n = 2^{24}$ commitments on random 128 bit values requires 11 seconds, resulting in 0.65 microseconds per commitment. When decommitting via sending the two secret shares for all n commitments, the implementation requires 5.4 seconds, 0.32 microseconds per decommitment. The same operation performed using batch decommit requires 4.6 seconds, 0.27 microseconds per decommitment. Furthermore, homomorphic operations can be performed extremely fast, simply requiring a few XOR instructions per operation, requiring less than 1 second for n operations.

We also benchmark our implementation in the 100Mbps WAN setting with a 80ms round trip latency. For $n = 2^{24}$, committing requires 39 seconds, 2.3 microseconds per commitment. Decommiting via sending both secret shares required 106 seconds, 6.2 microseconds per decommit, while the more communication efficient batch decommitment method required 30 seconds, 1.8 microseconds per decommitment.

bit size	Network	Commit	Decommit	Batch Decommit
128	10Gbps LAN	0.65	0.32	0.27
	1Gbps LAN	1.2	2.0	0.79
	100Mbps WAN	2.3	6.2	1.8
1	10Gbps LAN	0.07	0.12	0.13
	1Gbps LAN	0.22	0.56	0.57
	100Mbps WAN	0.56	1.5	1.6

Fig. 3. Amortized running times in microseconds (μs) for $n = 2^{24}$ commitments.

When performing bit commitments in the LAN setting, $n = 2^{24}$ commitments to random bits requires 1.3 seconds and decommitting requires 2 seconds. Interestingly, batch decommitting is slightly slower, requiring 2.3 seconds to decommit. We attribute this to the reduction in communication being outweighed by the added computation that is required by batch decommit. Indeed, in the 100Mbps WAN setting where communication is more expensive, batch decommitting is faster, requiring 25 seconds as opposed to 26.

In terms of communication overhead, committing to a random 128 bit value requires 134 bits, with an added 128 overhead for chosen message. Decommitting requires 524 bits, while batch decommitting only requires 128 bits with a constant additive overhead of 20,960 bits, independent of the number of decommitments. Committing to bits requires 40 bits of communication and decommitting requires 80 bits. Batch decommitting only requires a single bit of communication with a constant additive overhead of 3,200 bits, again independent of the number of decommitments considered.

5 Tutorial

SplitCommit and its dependencies can be found at `https://github.com/AarhusCrypto/SplitCommit`. This project utilizes the cmake build systems and capable of integrating into larger build systems with minimal overhead. Apart from cmake and other basic utilities, the implementation builds on the libOTe [Rin] library which requires the C++ utility library Boost [boo] and cryptography libary Miracl [mir]. For complete details on how to build these utilities, we refer the reader to the readme file contained in the git repository.

5.1 Random Commitments

We now turn to giving a basic overview of how to use the SplitCommit library by providing minimal code snippets. First, let us consider the case of committing to `num_commits` random commitments. For a given bit size, the primary class can be constructed as

```
#include "split-commit/split-commit-rec.h"
#include "split-commit/split-commit-snd.h"
...
SplitCommitSender commit_snd(input_bit_size);
SplitCommitReceiver commit_rec(input_bit_size);
```

The commitments themselves are held in a vector like class called `BYTEArrayVector`. Due to the nature of the commitment, the sender must have two such containers, one to hold each share of the commitment. The receiver will then hold some combination of these two shares in their own `BYTEArrayVector`.

```
std::array<BYTEArrayVector, 2> send_commit_shares{
   BYTEArrayVector(num_commits, commit_snd.cword_bytes),
   BYTEArrayVector(num_commits, commit_snd.cword_bytes)
};

BYTEArrayVector rec_commit_shares(num_commits, commit_rec.cword_bytes);
```

To fill these containers with the random commitments, the seed OTs for the `SplitCommit*` objects must be set. To set the sender's class, we must first construct a random number generator `osuCrypto::PRNG send_rnd` and a communication socket `osuCrypto::Channel send_channel`.

```
commit_snd.ComputeAndSetSeedOTs(send_rnd, send_channel);
```

The analogous operation must be applied to the `commit_rec` object in another thread/program. Once seed OTs are set, the `SplitCommit*` objects can repeatedly be used to commit and decommit to many values. For example, the sender calls

```
commit_snd.Commit(send_commit_shares, send_channel);
...
commit_snd.Decommit(send_commit_shares, send_channel);
```

to interactively commits and decommits to `num_commits` random values and stores the resulting commitment data in `send_commit_shares`. It is important to note that these function calls results in two way communication across the `send_channel` and as such the receiver object `commit_rec` must make the analogous function calls in a different thread/program.

After the call to `Commit`, `send_commit_shares` holds two containers, where the i'th rows hold a 2 out of 2 XOR secret sharing of the committed codeword. The first `input_bit_size` bits of which is the actually committed value while the remaining bits contain the codeword parity information. For the case where `input_bit_size` = 128, there is a utility function called `XOR_128` for computing the random value that is committed to.

```
std::vector<uint8_t> value(input_bit_size / 8);
XOR_128(value.data(), send_commit_shares[0][i], send_commit_shares[1][i]);
```

Concurrently to the sender calling `Commit` and `Decommit`, the receiver must call

```
if (commit_rec.Commit(rec_commit_shares, rec_rnd, rec_channel) == false) {
   throw std::runtime_error("bad commitment check, other party tried to cheat.");
}
...
BYTEArrayVector result_values(num_commits, commit_rec.msg_bytes);
if (commit_rec.Decommit(rec_commit_shares, result_values, rec_channel) == false) ↩
    {
   throw std::runtime_error("bad decommitment check, other party tried to cheat.")↩
       ;
}
```

Note that the receiver is required to check the return values of these function calls to ensure both the commitments and decommitments were correctly performed.

In addition to the `Decommit` function shown above, the library provides a more efficient mechanism for decommitting known as batch decommit. This method send roughly one fourth the amount of data over the network. The disadvantage of this approach is that it requires three rounds of interaction as opposed to a single round and performs additional computation. As already mentioned in Section 2.1, with some modifications to the library, these additional rounds could be eliminated using the Fiat-Shamir heuristic.

5.2 XOR Homomorphic Operations

Now we turn our attention to performing homomorphic operations on the committed values. Any subset of the committed values can be combined simply by XOR-ing their commitments together. For the common case of `input_bit_size` = 128 the library provides the utility function `XOR_CodeWords` to combine commitments. For example, the sender can create a new decommitment to the XOR of the i'th and j'th commitment as follows

```
std::array<BYTEArrayVector, 2> new_decommit{
   BYTEArrayVector(1, send_commit_shares.cword_bytes),
   BYTEArrayVector(1, send_commit_shares.cword_bytes)
};

XOR_CodeWords(new_decommit[0][0], send_commit_shares[0][i], send_commit_shares[0][↩
    j]);
XOR_CodeWords(new_decommit[1][0], send_commit_shares[1][i], send_commit_shares[1][↩
    j]);
```

The receiver then must perform the analogous operation as follows

```
BYTEArrayVector new_decommit(1, commit_rec.cword_bytes);
XOR_CodeWords(new_decommit[0], rec_commit_shares[i], rec_commit_shares[j]);
```

Note that the sender must XOR together both of the shares that they hold while the receiver only performs a single XOR. These new commitments can then be decommitted as shown above.

5.3 Bit Commitment

For the case of `input_bit_size` = 1, the user must manually compute the sender's committed value as

```
uint8_t value = (send_commit_shares[0][i][0] ^ send_commit_shares[1][i][0]) & 1;
```

in lieu of the `XOR_128` function. In addition, when performing homomorphic operations on the commitments, the function `XOR_BitCodeWords` can be used. Finally, the values reported to the receiver in the `Decommit` and `BatchDecommit` function calls are packed. To obtain the value at a specific position, the utility function `GetBit` returns the value of the bit at a given index. Note that due to the bits being packed in the result container `result_values`, it should be initialized as

```
 BYTEArrayVector result_values(BITS_TO_BYTES(num_commits), 1);
```

5.4 Chosen Message

To use this library with chosen messages, additional work is required by the user. The central idea is that a commitment to a random value can be translated to a commitment of a chosen value simply by publishing the XOR difference between the two values, effectively forming a one-time-pad encryption under the random value which is committed. The user must maintain this "translation" value for each commitment that it holds. To then perform homomorphic operations, the corresponding translation value must also be XOR-ed together, resulting in the translation value for resulting commitment. In particular, shown below are the operations for the sender to translate the 128 bit random commitments made above to chosen message commitments. The value contained in `translation2` when XORed with the decommitted value of `new_commit` will result in the XOR of `chosen_message0` and `chosen_message1`.

```
...

uint8_t* chosen_message0 = get_msg(0);
uint8_t* chosen_message1 = get_msg(1);

uint8_t translation0[16], translation1[16];

XOR_128(translation0,  commit_snd[1][0], commit_snd[0][0]);
XOR_128(translation0, translation0, chosen_message0);


XOR_128(translation1,  commit_snd[1][1], commit_snd[0][1]);
XOR_128(translation1, translation1, chosen_message1);

send_channel.send(translation0, 16);
send_channel.send(translation1, 16);

uint8_t translation2[16];
XOR_128(translation2, translation0, translation1);
```

```

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

References

AHMR15. Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 702–729. Springer, 2015.

boo. The Boost C++ Libraries. http://www.boost.org.

BR93. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73. ACM, 1993.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

CDD$^+$15. Ignacio Cascudo, Ivan Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, and Roberto Trifiletti. Additively homomorphic UC commitments with optimal amortized overhead. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 495–515. Springer, 2015.

CGH98. Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 209–218. ACM, 1998.

CJS14. Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 597–608. ACM, 2014.

FJNT15. Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. Tinylego: An interactive garbling scheme for maliciously secure two-party computation. *IACR Cryptology ePrint Archive*, 2015:309, 2015.

FJNT16. Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 542–565. Springer, 2016.

FS86. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

GIKW14. Juan A. Garay, Yuval Ishai, Ranjit Kumaresan, and Hoeteck Wee. On the complexity of UC commitments. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 677–694. Springer, 2014.

HM04. Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*, pages 58–76. Springer, 2004.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 2015.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

LP07. Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007.

mir. MIRACL Cryptographic SDK: Multiprecision Integer and Rational Arithmetic Cryptographic Library. `https://github.com/miracl/MIRACL`.

MR17. Payman Mohassel and Mike Rosulek. Non-interactive secure 2pc in the offline/online and batch settings. *IACR Cryptology ePrint Archive*, 2017:125, 2017.

NO09. Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

NST17. Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *24. Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society, February 26-March 1, 2017. To appear.

Rin. Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.