MPC and Security from Replication

Pille Pullonen

November 19, 2019
Security from Replication

- Cut-and-Choose
- MASCOT triple privacy amplification
How Else can we use Replication?

- Assume:
  - There are $n + 1$ parties who want to compute something
  - They know of a $n$-party passively secure protocol for MPC
    - Secure against one cheating party
    - That preserves input privacy until the output round
  - They are each willing to believe that at most one of the other parties might cheat
  - Can they compute together?
    - Yes! Just use the $n$-party protocol. Assuming everyone can give inputs to there

- What if the cheater is active?
- We can run $n + 1$ copies of the passive protocol each time leaving one party out
- At least one of the runs gives the correct result
- Need something to verify all runs before the opening
- How to do something like this more efficiently?
  - How to efficiently replicate the parties?
  - How to do the verification?
Combining MPC Schemes

- How to do the verification for the initial replication idea?
- We need some way to reliably check the values before they are opened
  - Could commit all values and open them
    - This may reveal more than we intend if we reveal the intermediate representation of the secrets
    - For SPDZ we looked at a way of achieving commitments from a modified secret sharing scheme
  - Could switch to some other MPC protocol to securely check equality of all outputs
    - E.g. convert the outputs to some verifiable secret sharing scheme for the final computation
- Switching between MPC protocols is often handy
  - Different operations are efficient or supported
  - Different guarantees provided
  - Different number of parties allowed
  - Different data structures supported
Garbled Circuits Reminder

- Two-party computation
- The garbler generates the garbled circuit by encrypting each gate and sends it to the evaluator
- The evaluator uses oblivious transfer to receive the relevant input keys from the garbler
- The evaluator decrypts the circuit gate-by-gate to come to the output
- Constant rounds of communication
- Works for binary circuits
- Can get active security from cut-and-choose
Three-Party Garbled Circuits with Replication

- Security against one malicious party
- Enhance the two party approach:
  - Split the garbler to two parties
  - The evaluator remains as one party
- The garblers agree on an initial randomness $r$
- They independently garble the circuit using this randomness
  - The garbled output should be the same
- Evaluator checks that it receives the same garbled circuits from both garblers
- The evaluator evaluates the circuit as usual
- Garbling protocol is secure against cheating evaluator
  - Breaking this means breaking the encryption
- Garbling protocol is also secure against an honest garbler
  - This is sufficient because if one of them misbehaves then the evaluator notices and aborts the protocol
Inputs for Three-Party GC

• Each of the three parties may have their own input
• Inputs $x_1, x_2$ of the garblers:
  • Both parties commit to all input wire labels
  • For each wire the order of the commitments is permuted
    • Opening them does not reveal if it is 0 or 1 label
  • Each party opens the commitments of their inputs
• Input $x_3$ of the evaluator:
  • Which garbler should you run OT with to get the keys?
  • We can avoid OT in the three party case
    • Let $x_3 = x_4 \oplus x_5$ be secret shared as $x_4$ and $x_5$
    • Instead of $f(x_1, x_2, x_3)$ compute (and garble)
      \[ f'(x_1, x_2, x_4, x_5) = f(x_1, x_2, x_4 \oplus x_5) \]
  • Evaluator sends $x_4$ to one garbler and $x_5$ to the other
  • The garblers open the respective commitments
  • The evaluator learns the order of the commitment permutations for its inputs from both garblers and checks that they are the same
  • Private because seeing $x_4$ or $x_5$ does not leak $x_3$
  • Correctness because commitments and their permutation order was verified
Compiler for MPC

• Take one protocol as input
• Give a changed protocol as an output
  • Passive security to active security
    • Passive protocols are easier to design
    • Passive security is easier to prove
    • E.g. GMW complier where each step of the semi-honest protocol is enhanced with zero-knowledge proof of correctness
    • E.g. the two-party passive garbled circuits to three-party active
  • From security with abort to complete fairness
• Compilers can have different generality
  • In terms of which protocols work as input
  • In terms of how many new requirements they introduce
Passive to Active with Replication

- Input: Any passively secure $n \geq 3$ party protocol
- Output: Actively secure $n$ party protocol
- Complier idea:
  - Each party, message and computation is replicated
  - Receiving parties check consistency
Compiler Details: Setup

- **Real parties** $P_1, \ldots, P_n$
  - e.g. servers really running, may be either honest or corrupted
  - These parties have the inputs
- **Virtual parties** $P_1, \ldots, P_n$
  - These are the parties for the passively secure protocol
  - Each virtual party $P_i = \{P_i, \ldots, P_{i+m-1}\}$ is played by $m$ real parties
- **Setup**: All parties in $P_i$ are given an initial randomness $r_i$ for that virtual party
- **Input a value** $x$ from $P_i$:
  - $P_i$ computes secret shares $x_j$ of $x$ as $x = \sum x_j$
  - For each $j$ : $P_i$ broadcasts $x_j$ to all parties in $P_j$
  - Each virtual party $P_j$ treats the values $x_j$ as their input
  - Virtual parties execute the input phase of the passively secure protocol with $x_j$ as input of $P_j$ to obtain $[x_j]$ and compute $[x] = \sum [x_j]$ to get the desired input in the representation of the passively secure protocol
    - Alternatively we can say that the initial function $f(x)$ has been replaced by $f'(x_1, \ldots, x_n) = f(\sum x_j)$ in the computation phase
Compiler Details: Computation

- Computation:
  - The virtual parties execute the protocol using randomness $r_i$
  - For each $P_i$ the real parties $P_i, \ldots, P_{i+m-1}$ execute the computations of that party and send all messages
  - For each message exchange from $P_i$ to $P_j$ each real party in $P_i$ sends the message to each real party in $P_j$
  - Each real party in $P_j$ checks that it received the same message from all parties in $P_i$
    - Abort if the messages differ (or some message is not received)
- Output: Output the values published in the computation phase
  - Assuming none of the message sending checks failed
Intuition of the Compiler Correctness

- Real parties in $\mathbb{P}_i$ share the initial randomness $r_i$
  - This is the only randomness they use in their computations
- Real parties in $\mathbb{P}_i$ all know the inputs $x_i$ of the virtual party
- If they receive the same messages $m$ and use randomness $r_i$ then they always compute the same messages
  - Each local computation is kind of deterministic function $f(m, x, r_i)$
- If some party cheats then we can see a difference between the messages of honest and corrupted parties
  - Sending wrong messages is the only cheating that might break the protocol
    - The only other thing to do is to do any local computations with the values received in the protocol
    - But passively secure protocol has to be secure against such semi-honest behaviour anyway
- The protocol fails if we detect the difference in the messages
- How to choose the virtual parties to ensure detection?
How to Create Virtual Parties?

- Real parties are distributed evenly
- Each virtual party is made up of $m$ real parties
- Need to ensure that at least one of these real parties is honest
- Therefore if we allow $t$ actively corrupted parties then $m > t$
- Most efficient if $m = t + 1$
- Each real party participates in $m$ virtual parties
- If $t$ real parties are corrupted, then adversary can see at most $tm$ values
- Need the passively secure protocol to be secure against $tm$ (or $t^2 + t$) corruptions
- $n \geq t^2 + t + 1$ means that $t < \frac{n}{2}$ we always have honest majority in the actively secure protocol
Example

- Three parties
- Additive secret sharing in a ring $\mathbb{Z}_{2^k}$
- Real parties $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$
- $\mathcal{P}_i = \{\mathcal{P}_{i-1}, \mathcal{P}_{i+1}\}$
- Setup: parties $\mathcal{P}_{i-1}, \mathcal{P}_{i+1}$ in $\mathcal{P}_i$ share initial randomness $r_i$
- Passive protocol is secure against two corrupted parties
- Want a compiled protocol that is secure against one actively corrupted party
  - $t = 1$
  - $t^2 + t = 2$
  - $n = 3 \geq t^2 + t + 1 = 1 + 1 + 1 = 3$
  - Hence these parameters are suitable for the compiler construction
Example: Passively Secure Protocol mod $2^k$

- **Addition** $[[x + y]] = [[x]] + [[y]] = (x_1 + y_1, \ldots, x_n + y_n)$
- **Adding a public value** $[[x + c]] = [[x]] + c = (x_1 + c, \ldots, x_n)$
- **Multiplication by a public constant** $[[c \cdot x]] = c \cdot [[x]] = (c \cdot x_1, \ldots, c \cdot x_n)$
- **Sharing a value** - Party chooses random $x_1, \ldots, x_{n-1}$ and computes $x_n = x - \sum_{i=1}^{n-1} x_i$, sends $x_i$ to party $i$
- **Publishing a value** - each party sends their share $x_i$, parties compute $x = \sum x_i$
- **Multiplication** $[[w]] = [[x \cdot y]] = [[x]] \cdot [[y]]$ with Beaver triple $[[a]], [[b]], [[c]]$ where $c = ab$
  - Compute $[[e]] = [[x]] - [[a]]$ and $[[d]] = [[y]] - [[b]]$
  - Publish $e = \sum e_i$ and $d = \sum d_i$
  - Compute $[[w]] = [[c]] + e \cdot [[b]] + d \cdot [[a]] + ed$
Example: Compiled Protocol

• Input:
  • Each $\mathcal{P}_i$ shares $x^{(i)} = x_1^{(i)} + x_2^{(i)} + x_3^{(i)}$
  • $x_j^{(i)}$ is broadcasted to $\mathcal{P}_j$
  • Each $\mathcal{P}_i$ receives $x_{i-1}^{(1)}, x_{i-1}^{(2)}, x_{i-1}^{(3)}$ on behalf of $\mathcal{P}_{i-1}$ and $x_{i+1}^{(1)}, x_{i+1}^{(2)}, x_{i+1}^{(3)}$ on behalf of $\mathcal{P}_{i+1}$
  • The $\mathcal{P}_j$ already have a valid sharing of the inputs $x^{(i)}$

• Local computations:
  • Parties $\mathcal{P}_{i-1}, \mathcal{P}_{i+1}$ in $\mathcal{P}_i$ carry out the computations of this party using shares $x_i$ and the shared randomness $r_i$

• Sending messages:
  • Parties $\mathcal{P}_{i-1}, \mathcal{P}_{i+1}$ in $\mathcal{P}_i$ send a message to $\mathcal{P}_j$ by sending it to $\mathcal{P}_{j-1}, \mathcal{P}_{j+1}$
  • Receiving parties $\mathcal{P}_{j-1}, \mathcal{P}_{j+1}$ check that they received the same message from $\mathcal{P}_{i-1}$ and $\mathcal{P}_{i+1}$

• Output: Each party outputs the published values
Sending Messages

- Parties $P_{i-1}, P_{i+1}$ in $P_i$ send a message to $P_j$ by sending it to $P_{j-1}, P_{j+1}$
  - For three parties $i = j + 1$ or $i = j - 1$ if $i \neq j$
  - Some messages are unnecessary because a real party sends a message to itself
- In general replicated messages are the main component that ensures security
- But do we need to check every message on the go?
  - If we don’t then errors may propagate in the computation
  - But it is still sufficient if we catch them before revealing anything important
  - Most messages, e.g. the multiplication openings in the example contain randomness
- In general, we don’t need to check every message if the passively secure protocol preserves privacy also for active adversaries up to the opening phase
- If we postpone the check then we can also postpone sending the replicated messages
Introducing the Brains

- For every $\mathbb{P}_i$ set one real party to be the brain $B_i$
  - Other parties will be kind of silent partners
- Sending messages:
  - Party $B_i$ in $\mathbb{P}_i$ sends a message $m$ to $\mathbb{P}_j$ by sending it to $\mathbb{P}_{j-1}, \mathbb{P}_{j+1}$
  - Silent parties $\mathbb{P}_i \setminus \{B_i\}$ simply compute and store the message $m$
  - Receiving parties $\mathbb{P}_{j-1}, \mathbb{P}_{j+1}$ store this message for later
- Opening
  - Need to verify the sent messages before opening anything important
  - The then send the opening messages and verify them
Sending Messages with Brains

- Each party, message and computation is replicated
- Receiving parties check consistency
- Postponing the check:
  - Store the messages
  - Hash the transcript
Transcript Verification

- Each silent party has stored the messages $x_1, \ldots, x_\ell$ that the brain has sent for a given virtual party
- Each receiving party has stored all messages received from the brain
- Before the Opening phase:
  - Each pair $P_i, P_j$ runs the check
  - How to implement the check?
    - All real parties in the sender side simply send their respective messages to the real parties in the receiver
- Three party case:
  - Let $h_{(i,j)}$ be the set of messages sent from $P_i$ to $P_j$
  - For $P_{i+1}$ where $P_i$ is the brain $P_i$ keeps the received messages $h_{i,i+1}$ and $h_{i-1,i+1}$
  - For $P_{i-1}$ $P_i$ keeps $h_{i-1,i}$, $h_{i,i-1}$, $h_{i-1,i+1}$.
  - For $P_{i-1}$ $h_{i+1,i-1}$ is not necessary because these messages are sent by $P_i$
Three Party Transcript Verification I

- \( \mathcal{P}_1 \): received \( h_{1,2}, h_{3,2}, h_{1,3} \), sent \( h_{3,1}, h_{3,2} \), discarded \( h_{2,3} \)
- \( \mathcal{P}_2 \): received \( h_{2,3}, h_{1,3}, h_{2,1} \), sent \( h_{1,2}, h_{1,3} \), discarded \( h_{3,1} \)
- \( \mathcal{P}_3 \): received \( h_{3,1}, h_{2,1}, h_{3,2} \) sent \( h_{2,3}, h_{2,1} \), discarded \( h_{1,2} \)
- Each \( h_{i,j} \) is kept by two different parties that need to verify it
- \( \mathcal{P}_i \) needs to locally check that the copies of \( h_{i-1,i+1} \) are equal
- Party \( \mathcal{P}_{i+1} \) can send \( h_{i,j} \) to the intended receivers \( \mathcal{P}_{j-1} \) and \( \mathcal{P}_{j+1} \). One of them is always either itself or the original sender of the message, so only one desired receiver remains:
  - If the original sender \( \mathcal{P}_{i-1} \) of the message was corrupted then both checkers are honest. If the sender corrupted some message then \( \mathcal{P}_{i+1} \) still computed it correctly so the checker notices the difference.
  - If the transcript sender \( \mathcal{P}_{i+1} \) is corrupted then the original sender \( \mathcal{P}_{i-1} \) was honest and an honest checker can notice the difference.
Three Party Transcript Verification II

• If the checker is corrupted then the transcripts matches but it can still call the check failed. But it could also deliberately fail the check independently of which algorithm we use.

• No privacy risk because the messages are sent to their intended receiver anyway
  • If this sending is problematic then the initial protocol without the brains has to be broken

• For three parties the verification is sufficient if the intended senders simply send their transcript to the intended receivers to check
  • No need for more complicated checks of equality, e.g. everyone committing to their transcript and then opening the commitments pairwise

• Note that we can actually combine the $h_{i,i-1}$ and $h_{i,i+1}$ because they are checked by the same real party
General Transcript Verification

- Let $h_{(i,j)}$ be the set of messages sent from $\mathbb{P}_i$ to $\mathbb{P}_j$
  - Each real silent party $\mathcal{P}_\ell$ in $\mathbb{P}_i$ keeps the computed messages $h_{(i,j),\ell}$. Whereas $h_{(i,j),\ell_1}$ and $h_{(i,j),\ell_2}$ may differ
  - For verification each $\mathcal{P}_\ell$ broadcasts his $h_{(i,j),\ell}$ to $\mathbb{P}_j$.
  - Initially $\mathcal{P}_\ell$ sent these messages during the protocol, nothing new is leaked if we send them during the verification

- There is at least one pair of honest real parties for each pair $\mathbb{P}_i$ and $\mathbb{P}_j$
  - For each send from $\mathbb{P}_i$ to $\mathbb{P}_j$ there exists some pair $\mathcal{P}_{i'} \in \mathbb{P}_i$ and $\mathcal{P}_{i'} \in \mathbb{P}_j$ such that both $\mathcal{P}_{i'}$ and $\mathcal{P}_{j'}$ are honest

- For each transcript $h_{i,j}$ verification:
  - If the brain was cheating then the honest receiver and honest transcript sender find the mismatch in the protocol
  - If all transcript senders are cheating then the brain was honest and the honest receiver checks the cheated transcripts against the honest stream of messages that it initially received
  - Hence, some pairwise check fails if there is any cheating
  - Hence, it suffices if all real parties simply send their transcripts to the intended receivers
Efficiency and Security of the Verification

- If all messages are sent as is then the overhead in communication is large (same as the protocol before introducing brains)
- Using a cryptographic collision resistant hash reduces communication to simply sending one hash for each pairwise verification
  - But requires more local computation and introduces a cryptographic assumption
  - Can reduce storage if we build a hash tree of the transcript instead of storing all messages
- Computing a random linear combination of all the messages reduces the communication to just one ring element per pairwise verification
  - Provides statistical security
  - But requires us to store all of the transcript because the random combination must be chosen later
Solving the Setup with Brains

- Setup: We need shared randomness $r_i$ for parties in $\mathbb{P}_i$.
- We can let the brain choose it.
- The brain multicasts it to the rest of $\mathbb{P}_i$.
- If the brain is honest then this is ok.
- If the brain is not honest then $r_i$ might be crafted to reveal information.
  - But the honest party in $\mathbb{P}_i$ still uses it as the randomness and computes values based on this.
  - The adversary already knows all secrets of $\mathbb{P}_i$ if the brain is corrupted.
  - So this is allowed as long as messages before the transcript verification don’t leak private information even if there is cheating.
    - This property has been called weak privacy/active privacy.
    - Not a very common definition.
    - But most passive MPC protocols have this property.
    - The use of brains has this precondition anyway.
The compiler is information theoretic

- Just uses secret sharing and replication based verification
- No new security assumptions introduced
  - Assuming the transcript verification does not introduce them
  - Verification could be implemented using a collision resistant hash function

The resulting actively secure protocol has the same security assumption as the initial passively secure one

- Notably if the passively secure protocol was information theoretically secure then so is the resulting active protocol
- But the number of corruptions that the active protocol tolerates is less than the passive protocol allows

WARNING: Proper security claims requires a proof of universal composability

- Have to build a simulator for the new construction
- Your simulator would use a simulator for the passively secure protocol to carry out the steps of the passively secure protocol
Efficiency of the Construction

- Simple echo broadcast (multicast) is still sufficient
- Each passive protocol message is replaced by $m$ messages
- Each share is stored in $m$ copies
- Computation overhead is also $m$ times if each real party plays the role of $m$ virtual parties
- Input round requires an extra layer of secret sharing
- Verification overhead depends on the choices
• Generating the triple
• Verifying the correctness of the triple
• Essentially could use the compiler on the preprocessing method to get a suitable preprocessing
• In the following we’ll look at more efficient ways to do preprocessing for three-party computation over $\mathbb{Z}_{2^k}$
  • Turning our focus to the way how the real parties see the protocols of the compiler
Additive Replicated Secret Sharing

- Consider the three party case with at most one corrupted party
- We have $x = x_1 + x_2 + x_3$
- Each party $P_i$ is given $x_{i-1}, x_{i+1}$
- $\llbracket x \rrbracket = ((x_2, x_3), (x_1, x_3), (x_1, x_2))$
- Opening a value:
  - $P_i$ sends $x_{i-1}$ to $P_{i-1}$ and $x_{i+1}$ to $P_{i+1}$
  - Each $P_i$ receives $x_i$ from both $P_{i+1}$ and $P_{i-1}$ and checks that it is the same
  - Each party outputs $x_1 + x_2 + x_3$
- Adding a constant
  $\llbracket x \rrbracket + c = ((x_2, x_3), (x_1 + c, x_3), (x_1 + c, x_2))$
- Addition
  $\llbracket x + y \rrbracket = ((x_2+y_2, x_3+y_3), (x_1+y_1, x_3+y_3), (x_1+y_1, x_2+y_3))$
- Generating a random share: $P_i$ picks $r_{i-1}$ and sends it to $P_{i+1}$, $\llbracket r \rrbracket = ((r_2, r_3), (r_1, r_3), (r_1, r_2))$
Additive Replicated Secret Sharing: Randomness

• Generating a random value:
  • Each party $P_i$ picks $r_i$ randomly
  • $P_i$ sends $r_i$ to $P_{i+1}$
  • $P_i$ receives $r_{i+1}$ from $P_{i-1}$
  • $P_i$ sets $(r_{i-1}, r_{i+1})$ as its shares of $[r]$

• If at most one party is corrupted then at least two $r_i$ are uniformly randomly chosen

• and the corrupted party can not fix the value of $r$
Optimistic Multiplication

Input: \([x], [y]\)
Output: \([xy]\)

- **Optimistic multiplication:**
  - Generate a random sharing \([r]\)
  - \(P_i\) computes \(u_{i+1} = x_{i+1}y_{i+1} + x_{i+1}y_{i-1} + x_{i-1}y_{i+1} + r_{i-1}\)
  - \(P_i\) sends \(u_{i+1}\) to \(P_{i-1}\) and receives \(u_{i-1}\) from \(P_{i+1}\)
  \([u] = ((u_2, u_3), (u_1, u_3), (u_1, u_2))\)
  \([xy] = [u] - [r]\)

- **Correctness:**
  - \(u_1 = x_1y_1 + x_1y_2 + x_2y_1 + r_2\)
  - \(u_2 = x_2y_2 + x_2y_3 + x_3y_2 + r_3\)
  - \(u_3 = x_3y_3 + x_3y_1 + x_1y_3 + r_1\)
  \[u - r = x_1y_1 + x_1y_2 + x_2y_1 + r_2 - r_1 + x_2y_2 + x_2y_3 + x_3y_2 + r_3 - r_2 + x_3y_3 + x_3y_1 + x_1y_3 + r_1 - r_3 = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3)\]

- Optimistic because it is possible to cheat in this protocol
Variants of Additive Replicated Secret Sharing

- The additive sharing can be over different data structures
  - All operations remain the same, but the shares and the secret value are from some ring or field
  - All operations are in that ring or field
- In the following we need:
  - \([x]_{2^k}\) for computations in a ring \(\mathbb{Z}_{2^k}\)
  - \([x]_p\) for computations in a field \(\mathbb{Z}_p\)
  - \([x]_\mathbb{Z}\) for the case when the shares are integers
    - Described in the following slides
Additive Replicated Secret Sharing over $\mathbb{Z}$

- Inputs $x \in \mathbb{Z}_{2^k}$ shared over integers with $k + \lambda$-bit initial shares
  - Security parameter $\lambda$ ensures that the shares of two different values are statistically close to each other
- Sharing $x$ to $\llbracket x \rrbracket_{\mathbb{Z}}$:
  - $P_i$ generates $x_1, x_2 \leftarrow \{0, \ldots, 2^{k+\lambda} - 1\}$, sets $x_3 = x - x_1 - x_2$. Sends $x_{j-1}, x_{j+1}$ to $P_j$
- Adding a constant, addition and multiplication with a constant are the same as for additive replicated secret sharing
  - BUT: The length of the shares may grow larger than $k + \lambda$-bits
  - Because all operations on shares are also over the integers
Optimistic Multiplication over \( \mathbb{Z} \)

**Input:** \([x]_{\mathbb{Z}}, [y]_{\mathbb{Z}}\)

**Output:** \([xy]_{\mathbb{Z}}\)

- Generate a random sharing \([r]_{\mathbb{Z}}\) where
  \(r_i \in \{0, \ldots, 2^{2\lceil \log B \rceil + \lambda + 2} - 1\}\)
  - \(B\) is a bound on the shares \(x_i, y_i \leq B\)
  - \(r_i\) are chosen at least \(\lambda\) bits longer than any \(x_i y_j\) multiplication

- \(\mathcal{P}_i\) computes \(u_{i+1} = x_{i+1}y_{i+1} + x_{i+1}y_{i-1} + x_{i-1}y_{i+1} + r_{i-1}\)

- \(\mathcal{P}_i\) sends \(u_{i+1}\) to \(\mathcal{P}_{i-1}\) and receives \(u_{i-1}\) from \(\mathcal{P}_{i+1}\)

- Check \(|u_i| \leq 2^{2\lceil \log B \rceil + \lambda + 3}\)

- \([u] = ((u_2, u_3), (u_1, u_3), (u_1, u_2))\)

- \([xy] = [u] - [r]\)
Triple Generation with Replicated Sharing

- $\mathcal{P}_i$ generates $a_i, b_i \in \mathbb{Z}_{2^k}$ and shares them as $[a_i]_Z, [b_i]_Z$
- Parties compute $[a] = \sum [a_i], [b] = \sum [b_i]$
- Use optimistic multiplication to compute $[c]_Z = [a]_Z \cdot [b]_Z$
- Verification of the multiplication:
  - Optimistically generate another triple $[x]_p, [y]_p, [z]_p$
    - for prime $p > c$
  - Interpret $[a]_Z, [b]_Z, [c]_Z$ as triple in $\mathbb{Z}_p$ for the verification with sacrifice:
    - Pick a random $[r]_p$ and open to $r$
    - $[e]_p = r[x] + [a]$
    - $[d]_p = [y] + [b]$
    - Open $e$ and $d$
    - Compute $[h]_p = ed - rd[x] - e[y] + r[z] - [c]$
    - Open $h$ and verify $h = 0$
- Set $[a]_{2^k}, [b]_{2^k}, [c]_{2^k}$ by locally reducing the shares of $[a]_Z, [b]_Z, [c]_Z$ to the correct size $\mod 2^k$
Correctness of Triple Generation

- Interpret \([a]_\mathbb{Z}, [b]_\mathbb{Z}, [c]_\mathbb{Z}\) as triple in \(\mathbb{Z}_p\)
  - We just treat \(a_i, b_i, c_i\) as values in \(\mathbb{Z}_p\)
- Multiplicative relation:
  - \(ab = c\) over the integers means \(ab = c \mod p\)
  - also \((a_1 + a_2 + a_3)(b_1 + b_2 + b_3) = (c_1 + c_2 + c_3)\)
  - \(p > c\) hence \(p > a, b\) and \(ab = c \mod p\) implies \(ab = c\) over the integers
  - Hence triple verification in \(\mathbb{Z}_p\) is usable
- Verification with sacrifice is the same as for the SPDZ case
  - All computation are in a field \(\mathbb{Z}_p\)
  - Probability of cheating is the probability of choosing a right \(r\), hence \(\frac{1}{p}\)
    - Likely need a \(p\) significantly bigger than the shares
- Final reducing:
  - We have \((a_1 + a_2 + a_3)(b_1 + b_2 + b_3) = (c_1 + c_2 + c_3)\) holding over the integers
  - Hence \((a_1 + a_2 + a_3)(b_1 + b_2 + b_3) = (c_1 + c_2 + c_3) \mod 2^k\)
  - Taking the modulo operations \(a_i \mod 2^k, b_i \mod 2^k, c_i \mod 2^k\) of the shares does not invalidate this property
So we have a way to generate and verify triples $[a]_{2^k}, [b]_{2^k}, [c]_{2^k}$

Verification requires generating an extra triple $[x]_p, [y]_p, [z]_p$

Essentially generate two triples and discard one

Can we do it more efficiently?

Yes!

Generate a set of triples $[a_i]_Z, [b_i]_Z, [c_i]_Z$ optimistically

Combine the triples so that we have one valid triple

Verify this triple with the check with sacrificing

Need one $[x]_p, [y]_p, [z]_p$ for the whole set of real triples

But how to do the combination?

We need some polynomial arithmetic for that
Polynomials

- \( f(x) = \sum_{i=0}^{t} a_i x^i \)
- Usually defined by the coefficients \( a_0, \ldots, a_t \)
- But you can define them by evaluation and value points \((x_j, y_j)\) instead where \( f(x_j) = y_j \)
- Degree \( t \) polynomial is uniquely defined by \( t + 1 \) points in either representation
- It is possible to translate between the representations
  - Either by evaluating the polynomial
  - or by interpolation
- The equation \( f(x) = \sum_{i=0}^{t} a_i x^i = 0 \) has at most \( t \) different solutions
- It is possible to do arithmetic with polynomials,
- These properties also holds when \( a_i \) at in some finite data structure, e.g. \( \mathbb{Z}_{2^k} \)
Polynomial Interpolation and Evaluation

- It is straightforward to evaluate polynomials in form
  \[ f(x) = \sum_{i=0}^{t} a_i x^i \]
- What about the case when polynomial is defined by \((j, f(j))\)?
- In principle we can interpolate to \(\sum_{i=0}^{t} a_i x^i\) representation and then evaluate
- Define \(\delta_i^N(x)\)
  \[ \delta_i^N(x) := \prod_{j=1, j \neq i}^{N} \frac{x - j}{i - j} \]
- Interpolation+evaluation can be written as one formula based on the Lagrange interpolation
  \[ f(z) := \sum_{j=1}^{t+1} \left( \delta_j^{t+1}(z) \cdot f(j) \right) \]
• Assume that the polynomial is defined by \((j, \lceil f(j) \rceil)\)

• Then

\[
\lfloor f(z) \rfloor := \sum_{j=1}^{t+1} \left( \delta_{j}^{t+1}(z) \cdot \lceil f(j) \rceil \right)
\]

can be computed locally by each party

• \(\delta_{j}^{t+1}(z)\) is a public value

• So the expression is just a linear combination of shares
Idea of Polynomial-based Verification

- We define $f(x)$ and $g(x)$ as polynomials of degree $N - 1$
  - They are uniquely determined by our triples as $f(i) = a_i$ and $g(i) = b_i$ for $i \in 1, \ldots, N$
- $h(x) = g(x)f(x)$ is a degree $2N - 2$ polynomial
  - Need $2N - 1$ points to uniquely define it
- We set $h(x)$ so that $h(x) = g(x)f(x)$ should hold if $c_i = a_i b_i$
  - First $i \in \{1, \ldots, N\}$ points: $h(i) = c_i = a_i b_i = f(i)g(i)$
  - For $i \in \{N + 1, \ldots, 2N - 1\}$ we compute $h(i) = f(i)g(i)$ by evaluating $f(i)$ and $g(i)$ and computing the multiplication
- If $h(x) \neq g(x)f(x)$ then
  - $h(x)$ and $g(x)h(x)$ have at most $2N - 2$ shared points
    - because $2N - 1$ shared points would already define the same $2N - 2$ degree polynomial, meaning that $h(x) = g(x)f(x)$
- Verify $h(x) = g(x)f(x)$ by evaluating both sides at a random point $z$
  - If $h(x) = g(x)f(x)$ then trivially $h(z) = g(z)f(z)$
  - Probability of choosing random point $z$ that is the shared point such that $h(z) = g(z)f(z)$ if $h(x) \neq g(x)f(x)$ is $\frac{2N-2}{p}$
Batch Verification - Fixing the Polynomials

Input: Set of triples \([a_i]_Z, [b_i]_Z, [c_i]_Z\), interpreted as \([a_i]_p, [b_i]_p, [c_i]_p\)

• Combine the triples to check them at once:
  • For \(i \in \{1, \ldots, N\}\), define \([f(i)] := [a_i]\) and \([g(i)] := [b_i]\).
  • For \(i \in \{N+1, \ldots, 2N-1\}\), evaluate the polynomials at point \(i\)
    \[
    [f(i)] := \sum_{j=1}^{N} (\delta^N_j(i) \cdot [a_j]), \text{ and}\n    [g(i)] := \sum_{j=1}^{N} (\delta^N_j(i) \cdot [b_j])
    \]
  • For \(i \in \{1, \ldots, N\}\), define \([h(i)] := [c_i]\).
  • For \(i \in \{N+1, \ldots, 2N-1\}\), compute \([h(i)] = [f(i)] \cdot [g(i)]\) optimistically.
  • Generate random \([z]_p\) and open \(z\).
Batch Verification - Verifying the Polynomials

Last slide: randomness $z$, hopefully $\left[ h(x) \right] = \left[ f(x) \right] \cdot \left[ g(x) \right]$

- Evaluate the polynomials on the random point $z$

$$\left[ \alpha \right] = \left[ f(z) \right] := \sum_{j=1}^{N} \left( \delta_j^N(z) \cdot \left[ f(j) \right] \right), \text{ and}$$

$$\left[ \beta \right] = \left[ g(z) \right] := \sum_{j=1}^{N} \left( \delta_j^N(z) \cdot \left[ g(j) \right] \right), \text{ and}$$

$$\left[ \gamma \right] = \left[ h(z) \right] := \sum_{j=1}^{2N-1} \left( \delta_j^{2N-1}(z) \cdot \left[ h(j) \right] \right)$$

- Run SacrificeCheck ($\left[ \alpha \right], \left[ \beta \right], \left[ \gamma \right]$).
  - Outputs true if $\alpha \beta = \gamma \mod p$

- Allows to sacrifice only one triple to verify many
Batch Verification Analysis

• From the polynomial idea we know that the probability of choosing a random value $z$ that makes $\gamma = \alpha \beta$ if any of the input triples was incorrect is $\frac{2N-2}{p}$

• The probability that the verification with sacrificing passes if $\gamma \neq \alpha \beta$ is $\frac{1}{p}$
• Generate a set of triples independently:
  • \( \mathcal{P}_i \) generates \( a_i, b_i \in \mathbb{Z}_{2^k} \) and shares them as \([a_i]_{\mathbb{Z}}, [b_i]_{\mathbb{Z}}\)
  • Parties compute \([a] = \sum [a_i], [b] = \sum [b_i]\)
  • Use optimistic multiplication to compute \([c]_{\mathbb{Z}} = [a]_{\mathbb{Z}} \cdot [b]_{\mathbb{Z}}\)

• Interpret the triples as triples in \(\mathbb{Z}_p\) and do batch verification

• If the verification succeeds then locally reduce
  \([c]_{\mathbb{Z}} = [a]_{\mathbb{Z}} \cdot [b]_{\mathbb{Z}}\) to \([c]_{2^k} = [a]_{2^k} \cdot [b]_{2^k}\)
Yet Another Share Representation

- Additive replicated secret sharing with redundant shares:
  - Values are $x \in \mathbb{Z}_{2^k}$
  - Shares as $x_i \in \mathbb{Z}_{2^k+\lambda}$, denoted as $[x]_{2^k+\lambda}$
  - All operations are as for regular additive replicates sharing

- Can be used to check the multiplicative property without computing in a finite field
Share Conversion

- Convert $[x]_{2^k}$ to $[x]_{2^k+\lambda}$:
  - No actual conversion, each party just computes with their shares as in $\mathbb{Z}_{2^k+\lambda}$
  - This means that likely $\sum x_i \mod 2^{k+\lambda} \neq x$
    - Either $\sum x_i \mod 2^{k+\lambda} = x$ or $\sum x_i \mod 2^{k+\lambda} = x + 2^k$
    - Since our interpretation is that $x \in \mathbb{Z}_{2^k}$ then these two are still suitable representations

- Convert $[x]_{2^k+\lambda}$ to $[x]_{2^k}$:
  - Each party locally reduces their share $x_i \mod 2^k$
  - The shared value is preserved because $2^k$ divides $2^{k+\lambda}$
Input $[x]_{2k}, [y]_{2k}, [z]_{2k}$ generated with optimistic multiplication

Output True if $z = xy \mod 2^k$

• Verification:
  • The parties convert the input shares $([x], [y], [z])$ into $([x]_{k,\lambda}, [y]_{k,\lambda}, [z]_{k,\lambda})$.
  • The parties generate a random $[a]_{k,\lambda}$ and execute an optimistic multiplication with $([a]_{k,\lambda}, [y]_{k,\lambda})$ to get $[c]_{k,\lambda}$.
  • The parties jointly generate a random $r \in \mathbb{Z}_{2^\lambda}$.
  • The parties reveal $[e]_{k,\lambda} = r[x]_{k,\lambda} + [a]_{k,\lambda}$.
  • Check $r[z]_{k,\lambda} + [c]_{k,\lambda} - e[y]_{k,\lambda} = 0$

• Similar ideas are used in SPD$\mathbb{Z}_{2k}$ for sharing and MASCOT for triple verification with correlated triples

• Can batch many pairwise sacrifices to use one $r$
Correctness of the SPDZ-like Verification

- **Correctness:**
  - \( r(xy) + ay - (rx + a)y = 0 \)

- **Security:**
  - Both optimistic multiplications can introduce errors
  - Assume \( z = xy + e_z \) and \( c = ay + e_c \) with errors, where \( e_z \neq 0 \mod 2^k \)
  - \( r(xy + e_z) + ay + e_c - (rx + a)y = re_z + e_c \mod 2^{k+\lambda} \)
  - \( r \) is uniformly random in \( \lambda \)-bit number
  - \( re_z \in \mathbb{Z}_{2^{k+\lambda}} \) is uniform in a set of at least \( 2^\lambda \)
  - \( e_c \) is chosen before \( r \) is sampled
  - Hence choosing right \( r \) to cheat the test can happen with at most \( 2^{-\lambda} \) probability
Verifiable Secret Sharing Scheme

- Secret sharing scheme
- Correctness of the shares can be checked
- If everything or sufficiently big subset is correct then we can open
- Usually needs broadcast at some point
  - Detectable broadcast is sufficient
    - Either everyone receives the message or everyone aborts
    - Unless we need to guarantee termination
- Not efficient for full MPC protocol but crucial for some steps
  - We’ll look at a compiler from security with abort to complete fairness
Compiler From Abort to Fair

- Security with abort
  - The adversary sees the output and can decide if the honest parties receive it or not
- Complete fairness
  - The adversary may abort but without seeing the output
  - Essentially the best to hope for in many settings
    - It is hard to guarantee termination in most settings
    - So it is hard to guarantee output delivery
- Preconditions:
  - Security with Abort
  - Output is revealed in the last round of computations
  - Honest majority
  - Detectable broadcast
  - Success of the opening can be decided only based on the messages in the opening round *
- Full active security with guaranteed output delivery is possible:
  - Honest majority, and
  - Unconditionally secure broadcast with termination
From Abort to Fair Protocol Idea

- Run the computation phase as is
  - Can abort but no outputs are revealed
- Opening round:
  - Let $d_{i,j}$ be the message of the output round sent from $\mathcal{P}_i$ to $\mathcal{P}_j$
  - Parties use VSS to secret share $d_{i,j}$
  - Parties use $d_{i,j}$ to check if the protocol should abort
  - All parties use detectable broadcast to publish if they aborted or not
  - If none of the parties aborted then all VSS shares of $d_{i,j}$ are sent to $\mathcal{P}_j$
  - Parties apply the reconstruction algorithm to learn $d_{i,j}$
    - The number of honest parties has to be such that successful opening is guaranteed if the protocol succeeded before
  - Parties follow the computations of the output round with $d_{i,j}$
• Compiler idea: Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. Ivan Damgård, Claudio Orlandi, and Mark Simkin. CRYPTO 2018

• Three party case: Use your Brain! Arithmetic 3PC For Any Modulus with Active Security Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. ePrint 2019

• GMW: How to play any mental game or A completeness theorem for protocols with honest majority. Oded Goldreich, Silvio Micali, and Avi Wigderson. ACM STOC 1987.