SPDZ

Pille Pullonen

November 12, 2019
Multiparty Computation from Somewhat Homomorphic Encryption.

Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias.
CRYPTO 2012
Main Characteristics of SPDZ

- Secure multiparty computation
- Additive secret sharing
- Any number of parties
- Preprocessing or online-offline paradigm
- Dishonest majority
- Active or malicious adversary
- Static adversary in the offline phase
- Adaptive adversary in the online phase
- Statistical security in the online phase
- Computational security in the offline phase
- Not identifying the cheater
- Security with abort
- Efficiency ($SPDZ = SPeDZ$)
Protocol Development

- BDOZa: 2011
- SPDZ: 2012
- SPDZ-2: 2013
- MASCOT: 2016
- Overdrive: 2018
- SPD\mathbb{Z}_{2k}: 2018
MPC with Preprocessing

- **Online phase**
  - Uses the private inputs
  - Computes the necessary functionality
  - Must be efficient

- **Offline phase**
  - Independent of the inputs
  - Independent of the desired computations
  - Can be used to prepare correlated randomness for the online phase
    - Random values
    - Beaver triples - \([a], [b] \) and \([c]\) where \(a\) and \(b\) are random and \(c = a \cdot b\)
  - Efficiency is less important
Additive Secret Sharing

- $[x] = (x_1, \ldots, x_n)$ where $x = \sum x_i$
- in SPDZ we have $x \in \mathbb{F}$
  - commonly $\mathbb{F}_p$ for some prime $p$
  - $x = \sum x_i \mod p$
Online Phase with Additive Secret Sharing

- 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 \([a], [b], [c]\)
  - Compute \([e] = [x] - [a]\) and \([d] = [y] - [b]\)
  - Publish \(e = \sum e_i\) and \(d = \sum d_i\)
  - Compute \([w] = [c] + e \cdot [b] + d \cdot [a] + ed\)
But active security?

- Need some way to authenticate or verify the sharing
- Message authentication codes (MAC)
- Can be used to authenticate either the share or the secret value
- Example:
  - Each party has a share $x_i$ and some tag $t_i$ for a MAC that other party can check
  - Does not scale - need a tag for each party
- Only works if the MAC is homomorphic
  - Because we need MAC-s on the computation
**SPDZ style MAC**

- Authenticate secret value not the separate shares
- \( \text{MAC}(x) = \alpha \cdot x \mod p \)
- Security - given \( x \) can we modify the message and create a valid MAC?
  - Can not see \( \alpha \) nor \( \text{MAC}(x) \), find \( e \neq 0 \) and \( e' \)
  - \( \text{MAC}(x + e) = \text{MAC}(x) + e' \)
  - Need to find \( e, e' \), we have \( \alpha e - e' = 0 \) for a successful attack
  - Probability \( \frac{1}{|F|} \) - equivalent to guessing the key
  - Need a big field

- Homomophic given \( \text{MAC}(x) \) and \( \text{MAC}(y) \)
  - \( \text{MAC}(x + y) = \text{MAC}(x) + \text{MAC}(y) = \alpha(x + y) \)
Share Representation with MAC

- Common key $\alpha$
- Additive shares of the secret and the MAC
  - $[x] = ((x_1, \text{MAC}(x)_1), \ldots, (x_n, \text{MAC}(x)_n))$
  - $x = \sum x_i$
  - $\text{MAC}(x) = \sum \text{MAC}(x)_i$
- Key $\alpha$ is additively shared $\langle \alpha \rangle = (\alpha_1, \ldots, \alpha_n)$ where $\alpha = \sum \alpha_i$
- How do we compute?
- How and when do we verify the MAC?
Computing with Additive Shares + MAC

- Addition $[x + y] = [x] + [y] = ((x_1 + y_1, \text{MAC}(x)_1 + \text{MAC}(y)_1), \ldots, (x_n + y_n, \text{MAC}(x)_n + \text{MAC}(y)_n)$
- Multiplication with a constant $[c \cdot x] = c \cdot [x] = ((c \cdot x_1, c \cdot \text{MAC}(x)_1), \ldots, (c \cdot x_n, c \cdot \text{MAC}(x)_n)$
- Adding a public value $[x + c] = [x] + c = ((x_1 + c, \text{MAC}(x)_1 + c \cdot \alpha_1), \ldots, (x_n, \text{MAC}(x)_n + c \cdot \alpha_n)$
- Multiplication with Beaver triples remains the same as before
- Sharing a value $x$ - we assume we have a random $[r]$ from preprocessing,
  - $[r]$ is published to the party
    - Can be done in preprocessing
    - Must be a verified opening
  - It computes and broadcast $x - r$
  - Players compute $[x] = [r] + (x - r)$
- Opening - how to verify the MAC?
Opening

- Exchange shares $x_i$ of $[x]$ to learn $x' = \sum x_i$
- Commit to shares $\alpha_i$ of the key
- Commit to shares $\text{MAC}(x)_i$ of the MAC
- Open the commitments and compute $\alpha' = \sum \alpha_i$
- Open the commitments and compute $\text{MAC}(x)' = \sum \text{MAC}(x)_i$
- Check if $\text{MAC}(x)' = \alpha' x'$
- The check passes if $x' = x$, meaning that the opening was correct, or the adversary broke the MAC and knows the key
  - The adversary sees $x$
  - but needs to find a modifier for the MAC before seeing $\alpha$ or $\text{MAC}(x)$
- This opening leaks the key $\alpha$ and therefore can be done only once. But we need two openings for each multiplication.
• Exchange shares $x_i$ of $[x]$ to learn $x' = \sum x_i$
• Each party locally computes $d_i = \alpha_i x' - \text{MAC}(x)_i$
• $d = \sum d_i = \alpha x' - \text{MAC}(x)$ need it to be 0.
• How to check?
  • Players commit to $d_i$, then open
  • Check the sum is 0
• This is equivalent to the MAC security
• Adversary has not seen $\alpha$ nor $\text{MAC}(x)$ before committing
  • Guessing a value to modify $d_i$ is the same as modifying ones MAC share
• No need to publish $\alpha$ - can continue computing with these shares
How many commitments do we need?

- Each sharing operation requires one opening
- Each multiplication requires two openings
- Each opening requires one commitment from all parties
- Number of commitments depends on the computation and number of parties
  - In short, many commitments
Commitments

• Could use any UC-secure commitment scheme

• Can also obtain a commitment similarly to the rest of the protocol:
  • Sharing with more MAC-s $\langle\langle x \rangle\rangle$
    • Each party defines a MAC key $\beta_i$
    • We define MAC-s for all parties $\text{MAC}(x, \beta_i) = \beta_i \cdot x$
    • We keep additive shares $[\text{MAC}(x, \beta_i)]$
    • Opening - send $x_i$ and $\text{MAC}(x, \beta_j)_i$ to party $j$
    • Each party can verify the opening on its own
  
  • Committing to $x$:
    • Obtain a random value $\langle\langle r \rangle\rangle$ from preprocessing
    • Open $r$ to the committer that broadcasts $x - r$
    • To open the commitment all parties open $r$ and compute $x$

• Still requires $O(n)$ storage for $\langle\langle r \rangle\rangle$ for each party
Removing Some Commitments

• Partial opening
  • Exchange $x_i$ and compute $x = \sum x_i$
  • Postpone the verification

• Errors propagate

• Only need to check when something crucial is opened
  • Beaver multiplication opens random values
  • Do we care if they are faulty?

• Batch check
  • Group the checks together - e.g. all multiplication checks at once
  • Group the true outputs
Batch Verification

- \([x_1], \ldots, [x_t]\) partially opened to \(x'_1, \ldots, x'_t\)
- We need that \(x_i = x'_i\) for all of these
- Do random linear combination,
  - choose \(e_1, \ldots, e_t\) randomly
  - compute \([y] = \sum e_i [x_i]\) and \(y' = \sum e_i x'_i\)
  - If \(x_i \neq x'_i\) then \(y = y'\) with at most \(\frac{1}{|F|}\) probability
- Can open \([y]\) with commitment (Opening v2)
  - \(d_i = MAC(y)_i - y' \cdot \alpha_i\)
  - Commit to \(d_i\) and open later
  - Check that \(\sum d_i = MAC(y) - y' \cdot \alpha = 0\)
- Number of commitments is independent of the computation
Cheating in Batch Verification

• The adversary cheats if \( \sum d_i = 0 \) and some \( x_i \neq x_i' \)

• We can say that the adversary modifies the MAC with \( e' \)

\[
\sum d_i = \sum MAC(y)_i - y' \cdot \alpha_i \\
= \sum (\sum e_j MAC(x_j)_i) - (\sum e_j x'_j) \cdot \alpha_i \\
= \sum \sum e_j MAC(x_j)_i - e_j x'_j \alpha_i \\
= \sum e_j \sum MAC(x_j)_i - x'_j \alpha_i = \sum e_j (\alpha x_j - \alpha x'_j) \\
= \alpha \sum e_j (x_j - x'_j) = e'
\]

• So \( \alpha = \frac{e'}{\sum_j e_j (x_j - x'_j)} \) if \( \sum_j e_j (x_j - x'_j) \neq 0 \), probability is \( \frac{1}{|F|} \)
Cheating in Batch Verification II

- What if \( \sum_j e_j(x_j - x'_j) = 0 \)
- Let’s look at this as a linear map \( f_{\delta x}(e) = \sum_{j=1}^t e_j \delta x_j \)
  - \( e = (e_1, \ldots, e_t) \)
  - at least some \( \delta x_j \neq 0 \)
  - \( f_{\delta x} : \mathbb{F}^t \rightarrow \mathbb{F} \)
  - From rank-nullity theorem

\[
\dim(\text{Ker}(f_{\delta x})) = \dim(\mathbb{F}^t) - \dim(\text{Image}(f_{\delta x})) = t - 1
\]

- \( e \) is chosen independently of \( x_i \) and \( x'_i \)
- Probability of choosing \( e \in \ker(f_{\delta x}) \) is \( \frac{|\mathbb{F}^{t-1}|}{|\mathbb{F}^t|} = \frac{1}{|\mathbb{F}|} \)
- Total probability of cheating is \( \frac{2}{|\mathbb{F}|} \)
Choosing the Random Vector

• $e_i$ must be chosen randomly and after the openings
• Choose random $\langle\langle u \rangle\rangle$, open it and set $e_i = u^i \mod p$
• $[y] = \sum e_i[x_i] = \sum u^i[x_i]$ and $y = \sum e_ix_i' = \sum u^ix_i$
• If $y = y'$ then $0 = y - y' = \sum_{i=1}^{t} (x_i - x_i') \cdot u^i$
• If $x_i \neq x_i'$ then check is falsely 0 if $u$ is a root of a non-zero polynomial $\sum_{i=1}^{t} (x_i - x_i') \cdot u^i$ of degree at most $t$
  • there are at most $t$ such roots
• Error probability becomes $\frac{t}{|F|}$
• Efficiency vs security tradeoff
  • Suitable batch size can be established depending on the application and the field size
SPDZ Execution

- Run secure setup to obtain the shares of the key $\alpha$
- Use preprocessing to obtain random values $\langle\langle u \rangle\rangle$, $[r]$ and Beaver triples $[a]$, $[b]$, $[c]$
- Run sharing protocol to obtain the inputs $[x]$
- Run the desired computations with partial openings
- Verify the partial openings of the computation phase with a batch check
  - Abort if the check fails
- Run the opening phase, verify the openings
  - Abort if the check fails
UC Security of the SPDZ Online Phase

- Universal Composability
  - Simulator
  - Environment
  - Ideal functionality

- Ideal functionality
  - Gets inputs $x$ from players
  - Computes the desired function
  - Sends output $y$ to the corrupted party
  - Sends output $y$ to honest parties if the adversary allows it
    - Adversary chooses to go forth or to abort
Simulator Idea

- Run the setup phase
- Generate preprocessing data and give corrupted parties part to the environment
  - Just run the preprocessing functionality because no private inputs are involved
- Run a simulated protocol with the environment
  - Run the honest party functionality with dummy inputs
  - Information theoretically secure so the transcript should not reveal the inputs anyway
- Give corrupted parties input to the ideal functionality
- Get the output from the ideal functionality
- Decide if honest parties get output, tell it to the functionality
Simulated Protocol

- **Input stage**
  - When a corrupted party sends \( c \) as the modifier of share \([r]\) then the input is \( x = c + r \). The simulator gives this to the ideal functionality.

- **Computation stage**
  - Just follow the protocol with dummy inputs for honest parties.

- **Output stage**
  - Get the output \( y \) from the ideal functionality.
  - The simulated protocol currently has simulated some \([y']\).
  - Adjust the shares of the honest party to get \([y]\) (simulator knows the MAC key from simulating setup).
  - Simulate the opening protocol with the environment.
  - If all the checks pass in the simulated protocol then the simulator inputs OK to the ideal functionality.
    - Otherwise it inputs ABORT.
How Good is this Simulation?

- Correct protocol transcript
- Correct output (same as from the ideal functionality)
- BUT the real protocol may not give the correct output
- Incorrect output only happens if the environment cheats with the MAC
- Statistical security
  - Simulation is perfect as long as the MAC is not broken
  - MAC is broken with some known probability
  - Breaking the MAC does not depend on computational power
- WARNING: full UC proof should consider more details
  - Timing of all the messages and execution
Possible Improvements of SPDZ Online Phase

- Communication complexity of the opening
- Broadcast requirement
Partial Opening with Less Communication

- All parties send their share $x_i$ to the first party
- The party computes $x = \sum x_i$
- The party broadcasts $x$
- Can this party break the protocol?
- No - it gets no additional power compared to lying about its share
- This has one broadcast vs everyone broadcasting their value
Isn’t Broadcast Inefficient?

- In general yes!
- SPDZ can use Echo Broadcast:
  - Party sends their message to all other parties
  - Each party forwards the received messages to all other parties
  - Each party checks that all messages it receives are the same
    - Abort if there is a mismatch
  - It may not terminate, but also SPDZ does not guarantee termination in other steps
- If many messages are opened together then instead of echoing all of them the parties can just send a hash of the messages (amortization)
Efficiency of SPDZ Online Phase

• Storage
  • Each secret value $x$ is represented by $x_i$ and $\text{MAC}(x)_i$
  • Need to store the precomputed values
  • For each $\langle\langle r \rangle\rangle$ each party stores $n$ MAC values. Need at least $n$ values
    • Each party commits at least one value to do an opening

• Computation
  • Each arithmetic operation requires constant local computation
  • For opening $\langle\langle r \rangle\rangle$ need to sum $n$ values. Need to open at least $n$ values to verify the commitments.

• Communication
  • Multiplication does 2 partial openings
  • Sharing and opening require 1 partial opening
  • Each party sends $\mathcal{O}(n)$ messages in partial opening (echo broadcast + opening to one party)
Other Variants of the Online Phase

- What if I use a small field?
  - For example binary $\mathbb{F}_2$
  - MAC security is $\frac{1}{|\mathbb{F}|}$
  - Use many MAC-s with different keys
    - Big overhead in storage
  - If you always do vector operations then you can authenticate the vector (MiniMAC protocol)
- What if finite field is not the best for your application?
• Rings $\mathbb{Z}_{2^k}$ are a more common data structure than $\mathbb{F}_p$

• $x \in \mathbb{Z}_{2^k}$, $x_i \in \mathbb{Z}_{2^{k+s}}$

• key $\alpha \in \mathbb{Z}_{2^{k+s}}$ and MAC $m = \alpha \cdot x \mod 2^{k+s}$

• Breaking the MAC: $m + e' = \alpha' \cdot (x + e) \mod 2^{k+s}$ with $e \neq 0 \mod 2^k$

• Batch checking also works but is more complicated to analyze

• Online computation protocols remain the same
The online phase assumes:
- Beaver triples
  - Need to ensure that $a$ and $b$ are random
  - Need to ensure that $c = a \cdot b$
- Random values
  - Sharing and MAC shares
  - Also required for the triples

How can they be prepared?
Somewhat Homomorphic Cryptosystem (SHE)

- Public key encryption with homomorphic properties
- $\text{Enc}_{pk}(a + b) = \text{Enc}_{pk}(a) + \text{Enc}_{pk}(b)$
- $\text{Enc}_{pk}(a \cdot b) = \text{Enc}_{pk}(a) \cdot \text{Enc}_{pk}(b)$
- Each operation adds some noise
- $\text{Dec}_{sk}(\text{Enc}_{pk}(a)) = a$ if there is not too much noise
- SPDZ assumes:
  - that any number of additions and one multiplication is allowed
  - The plaintexts are in a finite field and the homomorphism is in that field
  - Plaintexts are vectors - SIMD operations
  - Triple generation can be parallel
  - Distributed decryption is possible
  - For example the BGV scheme
Distributed Decryption

- Each party knows additive share $sk_i$ of the secret key
- No-one has the whole secret key $sk$
- Collaborative process to decrypt
  - Each party computes some value $d_i$ from $sk_i$ and $Enc(a)$
  - The parties publish $d_i$ and compute $a$ from them
- Semi-honest security (there can be an error in the result)
- SPDZ assumes this setup so that ciphertexts can be published without leaking the contents
Setup of the MAC Key

- Each party chooses $\alpha_i$, publishes $\text{Enc}_{pk}(\alpha_i)$
- Each party does a zero-knowledge proof of knowledge of $\alpha_i$
- Set $\text{Enc}_{pk}(\alpha) = \sum \text{Enc}_{pk}(\alpha_i)$
  - Needed to ensure that each party chooses $\alpha_i$ itself
  - E.g. does not pick $\text{Enc}_{pk}(\alpha_i)$ based on seeing $\text{Enc}_{pk}(\alpha_j)$
- Let’s call this $\text{EncRnd}$ with $\alpha_i$ and $\text{Enc}_{pk}(\alpha)$ as outputs
- $\text{EncRnd}$ can be used to initialize any random sharing
Adding MACs to Shared Values with SHE

- **Input:** $\text{Enc}_{pk}(x)$, $\text{Enc}_{pk}(\alpha)$, additive shares $\alpha_i$ of $\alpha$
  - For example, $x$ can be generated with EncRnd
- **Output:** additive shares of $\text{MAC}(x) = \alpha \cdot x$
- **MAC computation:**
  - Run the EncRnd to get $r_i$ and $\text{Enc}_{pk}(r)$
  - Compute $\text{Enc}_{pk}(m) = \text{Enc}_{pk}(x) \cdot \text{Enc}_{pk}(\alpha)$
  - Compute $\text{Enc}_{pk}(d) = \text{Enc}_{pk}(m) - \text{Enc}_{pk}(r)$
  - Run distributed decryption on $\text{Enc}_{pk}(d)$ to learn $d = \alpha x - r + e$ where $e$ is the error (if any)
  - Set the MAC as $\text{MAC}(x)_1 = r_1 + d$, $\text{MAC}(x)_i = r_i$
  - This is a correct MAC if $e = 0$ and there was no error
  - $\sum \text{MAC}(x)_i = d + \sum r_i = \alpha x - r + e + r = \alpha x + e$

- **MAC is secure against errors**
  - The MAC security definition
  - The adversary does not know $\text{MAC}(x)$ or $\alpha$ when choosing $e$
Additive Triple Generation with SHE

- Each player chooses $a_i$, $b_i$, $r_i$ at random
- Broadcast $\text{Enc}_{pk}(a_i), \text{Enc}_{pk}(b_i), \text{Enc}_{pk}(r_i)$
- Each party does a zero-knowledge proof of knowledge of $a_i$, $b_i$, $r_i$
  - ZK needed to ensure input independence
- $\text{Enc}_{pk}(a) = \sum \text{Enc}_{pk}(a_i)$
- $\text{Enc}_{pk}(b) = \sum \text{Enc}_{pk}(b_i)$
- $D = \text{Enc}_{pk}(a) \cdot \text{Enc}_{pk}(b)$
- Compute and decrypt $d = D_{sk}(D - \sum \text{Enc}_{pk}(r_i)) = ab - r + e$
- Set $c_1 = r_1 + d$, $c_i = r_i$ to define $[c]$
- Set $\text{Enc}(c) = \sum \text{Enc}_{pk}(r_i) + \text{Enc}(d)$
  - Need a fresh encryption of $c$ because $D$ already contains noise from the multiplication but another multiplication is needed to generate MAC-s
- Use $\text{Enc}(a)$, $\text{Enc}(b)$, $\text{Enc}(c)$ and $\text{Enc}(\alpha)$ to generate the MAC shares for $[a], [b], [c]$
Triple Verification (with Sacrificing)

- The ZK proof in the triple generation guarantees that the inputs $a_i, b_i, r_i$ are random but nothing guarantees that the MAC-s or the multiplication is computed correctly.
- Input: additive shares with MACs $[a], [b], [c]$
- Output: True if $a \cdot b = c$
- Verification:
  - Take another (possibly faulty) triple $[x], [y], [z]$
  - Choose a random $\langle\langle r\rangle\rangle$ and open (with verification) to all parties to learn $r$
  - Compute $[e] = r \cdot [a] - [x]$
  - Compute $[d] = [b] - [y]$
  - Partially open $e$ and $d$
  - Compute $[h] = r \cdot [c] - [z] - e \cdot [y] - d \cdot [x] - d \cdot e$
  - Partially open $h$
  - Fail if $h \neq 0$
  - Verify all partial openings (batch verification)
Correctness of Triple Verification

• If $c \neq a \cdot b$ but the verification succeeds

\[ h = r \cdot c - z - e \cdot y - d \cdot x - d \cdot e \]
\[ = r \cdot c - z - (ra - x) \cdot y - (b - y) \cdot x - (b - y) \cdot (ra - x) \]
\[ = r(c - ab) - z + yx = 0 \]

• This can hold for exactly one $r = \frac{z - yx}{c - ab}$

• Failure probability is $\frac{1}{|F|}$

• If any of the MAC-s is invalid then the verified opening fails
  • Probability of cheating in this phase is $\frac{1}{|F|}$

• Sacrificing one triple to check another ensures correctness of the MAC-s as well as $c = ab$

• Triple $[x], [y], [z]$ randomizes the initial triple so that the published values hide $[a], [b], [c]$
Batched Triple Verification

- **Input**: Set of additive shares with MACs $[a_i], [b_i], [c_i]$
- **Output**: True if $a_i \cdot b_i = c_i$
- **Verification**:
  - Take another (possibly faulty) set of triples $[x_i], [y_i], [z_i]$
  - Choose a random $\langle\langle r \rangle\rangle$ and open (with verification) to all parties to learn $r$
  - Compute $[e_i] = r \cdot [a_i] - [x_i]$
  - Compute $[d_i] = [b_i] - [y_i]$
  - Partially open $e_i$ and $d_i$
  - Compute $[h_i] = r \cdot [c_i] - [z_i] - e_i \cdot [y_i] - d_i \cdot [x_i] - d_i \cdot e_i$
  - Partially open $h_i$
  - Fail if $h_i \neq 0$
  - Verify all partial openings (batch verification)

- Same $r$ can be used for all triples in one batch, privacy of final triples is still ensured because each triple is randomized by different $[x_i], [y_i], [z_i]$ triple
Variants of Preprocessing

• Fully homomorphic encryption / somewhat homomorphic encryption
  • Works for the general case
  • Efficiency is questionable
  • Zero-knowledge is expensive
  • Encryption is expensive

• Oblivious transfer
  • Very natural for the two-party case
  • Gained popularity after OT-extensions became known
  • Used by MASCOT

• Additively homomorphic encryption
  • Or only additive operation used in SHE
  • Used by Overdrive and BDOZa
Multiparty Triples from Two-Party Multiplication

\[
[a \cdot b] = [a] \cdot [b] = (a_1 + \ldots + a_n) \cdot (b_1 + \ldots + b_n)
\]

- Each component \(a_i \cdot b_i\) is computed locally by party \(i\)
- Each component \(a_i \cdot b_j\) is computed with two-party multiplication, parties learn \((a_i b_j)_1\) and \((a_i b_j)_2\) respectively
- Then each party sums \(c_i = a_i b_i + \sum_j (a_i b_j)_1 + \sum_j (a_j b_i)_2\)
• Preprocessing from correlated OT
• Two party multiplication with OT:
  • First party knows $a$ and the second party knows $b = \sum 2^i B_i$
  • The parties run OT for each bit $B_i$:
    • First party always inputs $a$
    • Second party inputs $B_i$
    • First party learns random $x_i \in \mathbb{F}$
    • The second party learns $y_i = x_i + B_i \cdot a$
  • First party computes $x = -\sum 2^i x_i$
  • Second party computes
    \[ y = \sum 2^i y_i = \sum 2^i (x_i + B_i \cdot a) = -x + a \cdot b \]
  • $x + y = a \cdot b$
• What if the first party does not always input \(a\)?
  • Assume it inputs \(a + \delta\) for round \(\ell\)
  • Then
    \[
    y = \sum 2^i y_i = \sum 2^i (x_i + B_i \cdot a) + \delta \cdot B_\ell = -x + a \cdot b + \delta \cdot B_\ell
    \]
• Selective failure attack:
  • If any sort of correctness check succeeds then the first party knows that \(B_\ell = 0\)
Avoiding Selective Failure

- Using it for triple generation:
  - Privacy amplification - combine many triples to one
    - Generate $a, \vec{b} \in \mathbb{F}_m$ and $\vec{c} \in \mathbb{F}_m$ where $a \cdot \vec{b} = \vec{c}$
    - Sample random $\vec{r} \in \mathbb{F}_m$
    - Compute $b = \langle \vec{b}, \vec{r} \rangle$ and $c = \langle \vec{c}, \vec{r} \rangle$
    - $a \cdot b = a \cdot \sum r_i b_i = \sum r_i (ab_i) = \sum r_i c_i = c$
    - $a \cdot b = c$
  - Verify triple correctness with sacrificing
    - Can also use correlated triple $[[a]], [[y]], [[w]]$ where $w = ay$
    - This triple can be generated like $[[a]], [[b]], [[c]]$ but using different randomness $r$
  - Any leaking bit in the original triple is randomized by the non-leaking bits

- Using it for MAC generation:
  - Error can be found similarly to the batch verification - checking a random linear combination of the MACs
  - We include one random value shared value that is used only in this linear combination
Multiparty Triples with Privacy Amplification

- Every party $P_i$ samples random $a_i \in \mathbb{F}$ and $\vec{b}_i \in \mathbb{F}^m$
- Every pair of parties $P_i$ and $P_j$ computes:
  - Two party multiplication for $a_i \vec{b}_j$
  - Output is $\vec{c}_{(i,j),i}$ for party $i$ and $\vec{c}_{(i,j),j}$ for party $j$
- Each party sets $\vec{c}_i = a_i \vec{b}_i + \sum_{i \neq j} \vec{c}_{(i,j),i} + \vec{c}_{(j,i),i}$
- Do privacy amplification:
  - Sample two random vectors $\vec{r}, \hat{\vec{r}}$
  - Set $b_i = \langle \vec{r}, \vec{b}_i \rangle$
  - Set $\hat{b}_i = \langle \hat{\vec{r}}, \vec{b}_i \rangle$
  - Set $c_i = \langle \vec{r}, \vec{c}_i \rangle$
  - Set $\hat{c}_i = \langle \hat{\vec{r}}, \vec{c}_i \rangle$
- Generate MACs for additively shared $[a], [b], [c], [\hat{b}], [\hat{c}]$
  - Essentially all parties use their shares as inputs
- Run triple verification with sacrificing to get a valid triple $[a], [b], [c]$
Triple Verification with Correlated Triples

• Input: Triples $[a], [b], [c]$ and $[a], [\hat{b}], [\hat{c}]$
• Output: Valid triple $[a], [b], [c]$
• Sacrificing:
  • Open a random $\langle\langle r \rangle\rangle$ to get $r$
  • Compute $[p] = r \cdot [b] - [\hat{b}]$
  • Open $[p]$ to get $p$
  • Compute $[t] = r \cdot [c] - [\hat{c}] - p[a]$
  • Open $[t]$ to get $t$
    • Abort if $t \neq 0$
    • If everything is correct, then
      $$t = rc - \hat{c} - pa = rab - a\hat{b} - rab + a\hat{b} = 0$$
  • Verify all opened values
• Correlated triples are more efficient to verify and also more efficient to generate
Overdrive

• Preprocessing from additively homomorphic encryption
• Inspired by the BDOZa multiplication
• Concrete protocol makes use of the interesting properties of the BGV encryption scheme to avoid the proof of correctness of multiplication
• Going back to some ideas (ZK proof) of original SPDZ that were discarded in SPDZ-2
• Use the privacy amplification idea from MASCOT
  • Also the two-party to multiparty multiplication
  • Correlated triple verification
BDOZa Multiplication Idea

- Additively homomorphic encryption
  \[ \text{Enc}_{pk}(a + b) = \text{Enc}_{pk}(a) + \text{Enc}_{pk}(b) \]

- Two-party multiplication protocol for \( ab \):
  - First party encrypts \( \text{Enc}_{pk}(a) \) with its key and sends to the second party
  - Second party computes \( \text{Enc}_{pk}(c_1) = b \cdot \text{Enc}_{pk}(a) - \text{Enc}_{pk}(c_2) \) for some random \( c_2 \) and sends it back
  - First party decrypts to get \( c_1 \)
  - Output \( c_1, c_2 \)

- Need zero-knowledge for:
  - Correctness/knowledge of \( \text{Enc}_{pk}(a) \)
  - Correct computation of \( \text{Enc}_{pk}(c_1) \)
    - This is avoided by Overdrive using a special property of their encryption scheme in combination with the triple verification

- Can lift to multiparty multiplication the same as before
What Else Can Be Preprocessed?

• Square pairs
  • $[a], [a^2]$  
  • Less storage than triples  
  • Multiplication algorithm can also be simplified

• Shared bits
  • $[a], a \in \{0, 1\}$  
  • Useful for comparison, bit-decomposition

• Input randomizers
  • Input party learns $r$, all parties hold $[r]$  
  • Like doing the first step of the sharing protocol in the preprocessing phase

• Probably other things for specific operations
SPDZ summary

- **Online phase:**
  - Additive secret sharing in some finite field
  - Simple homomorphic MAC that is kept additively shared
  - Linear combinations of shares are done locally
  - Multiplication is done with Beaver triples

- **Offline phase:**
  - Generating random values with MACs is essentially multiplication of two shared values
  - Generating triples is essentially multiplying two random values
  - Need to verify that the MAC-s are correct and that the multiplicative property holds
  - Multiplication can be done with somewhat homomorphic encryption, additively homomorphic encryption, oblivious transfer
• SPDZ: Multiparty Computation from Somewhat Homomorphic Encryption. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. CRYPTO 2012


• Ivan Damgård’s lecture in Bar-Ilan Winter School 2015 https://www.youtube.com/watch?v=N80DV3Brds0

• SPDZ\textsubscript{2k}: efficient MPC mod for dishonest majority. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. CRYPTO 2018
References and Extra Materials II

- **MASCOT**: Faster malicious arithmetic secure computation with oblivious transfer. Marcel Keller, Emmanuela Orsini, and Peter Scholl. ACM CCS 2016.
- **MiniMAC**: Constant-overhead secure computation of boolean circuits using preprocessing. Ivan Damgård, Sarah Zakarias. TCC 2013.
- **BGV**: (Leveled) fully homomorphic encryption without bootstrapping. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. ITCS ACM 2012.
• **UC:** Universally Composable Security: A New Paradigm for Cryptographic Protocols. Ran Canetti. 
https://eprint.iacr.org/2000/067