CS-523 SMCompiler Report

Manon Michel, Tom Demont

I. Introduction

The goal of this project is to implement and apply a secure multi-party computation engine (SMC) in a semi-honest (passive) adversarial setting using the Python 3 programming language. We implement the necessary components for SMC on arithmetic circuits assuming the existence of a trusted third-party and additive secret sharing.

II. Threat Model

The threat model we are secure under is honest but curious parties. We rely on the fact that parties compute the correct circuit (they do not deviate from the protocol) and can at most try to deduce information on inputs.

Additive Secret Sharing benefits from strongest link security which implies that an adversary must corrupt all but one players in order to reconstruct the secret. Our threat model is therefore secure under passive adversaries. Notably we rely on the fact that parties compute the correct circuit. If a party is malicious it can only distort the final result but it can not determine the value of other parties secrets (unless there is only one honest party). Likewise re-using a secret can lead to leakage. We must finally consider (and will discuss in the last part) the potential inference about secret from observed result which the scheme does not protect from.

In addition to this, if a party does not participate the others will wait on this party (to receive shares for example) which can slow down or lead to a non-termination of the scheme.

III. Implementation details

1) prime_gen.py: We implemented prime generation, `prime_gen`, using the Miller-Rabin Primality Test (as seen in COM-401). The test, `is_prime`, checks primality for an integer n under a parameter k which influences the probability of success of the test as follows: $P(\text{output n maybe prime} \mid \text{n composite}) \leq 4^{-k}$. This was not necessary but we did it for completeness as having a prime modulus is useful in this scheme.

2) secret_sharing.py: In order to implement secret sharing, we pre-compute a field value for our framework, q, of 3525679 using `prime_gen` with $k = 32$.

3) expression.py: We implemented a method to know which operands are scalars and which are not as this will change how we process the operation in *smc_party.py*: `has_scalar_operand`.

4) ttp.py: The Trusted Third Party (ttp) generates random values for the Beaver triplet multiplication scheme. A new triplet will be created if being called for a new operation, otherwise we obtain the shares for the stored triplet assigned to that operation id. The class `BeaverTriplet` represents a beaver triplet, it notably assigns the a, b, c values such that a, b are sampled smaller than $\sqrt{q}$ to ensure that $a \cdot b = c \in \mathbb{Z}_q$. Shares for a, b, c are then generated as seen in $secret\_sharing.py$.

5) smc_party.py: To determine which party should perform unique operations, we designate the first participant as the 'aggregating client'. Finally, in order to improve performance, we cache the shares and beaver triplets needed for the scheme.

IV. Performance Evaluation

To evaluate the performances, we made use of the module `pytest-benchmark`. Details of benchmarks can be found in the `.benchmarks` folder (also including nice visual histograms).

To correctly evaluate the variable's influence, we kept through each of these experiments the same number of $Secret$s in the system: 2, the minimum for a non-trivial application. For the evolution on number of parties, we kept constant the number of operations: an addition of both secrets with increasing number of parties. On the other hand, when varying the number of operations, we kept 3 constant parties, 2 having secrets and 1 having none. We decided to do so to eventually observe better the influence of variables as we were afraid keeping only one to one communication could mask increasing shape of communication costs. We are aware that this had a negative influence on the run time of the single cored machine, as every new party has to share a core with the other processes, but as this is kept constant over each experiment, we could still extract interesting statistics.

1) Communication cost evaluation: The communication cost evaluation was run by augmenting the server: on every `GET` request, the server adds to a counter the number of bytes of the given response. Once the protocol is finished, our tests retrieve from the server the number of bytes it sent to parties. We did not look at bytes `POST`ed by the parties as we considered a "peer to peer" communication model where we evaluate the total volume it would cost to have all our communications on P2P links, which is a relevant model for our application where, in deployment, we would like to depend less on trusted third parties.

An interesting aspect showed by our communication costs results is the cache of $Share$ and $BeaverTriplet$ we implemented at the party level. This results in constant communication overhead with increasing number of additions. Finally, we observed no variation over multiple runs, which makes sense as we expect sharing elements of the same size at each run. Also no variation on both 8cores arm64 and 1core x86_64 machines, which also sounds reasonable.

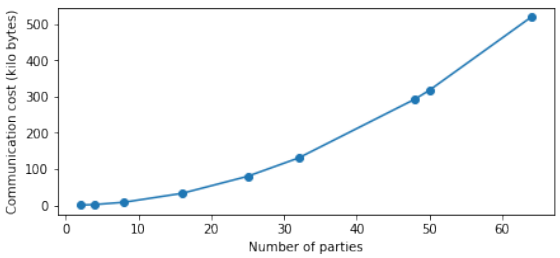

Fig. 1: *Communication cost evolution with nb of parties*

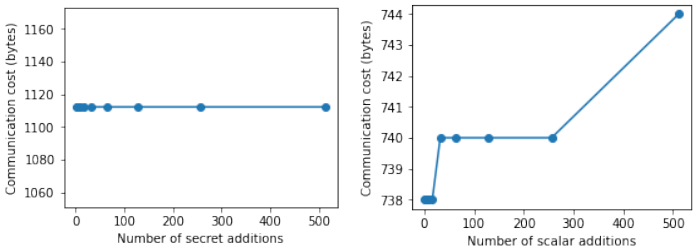

Fig. 2: *Communication cost evolution with nb of secret/scalar additions*

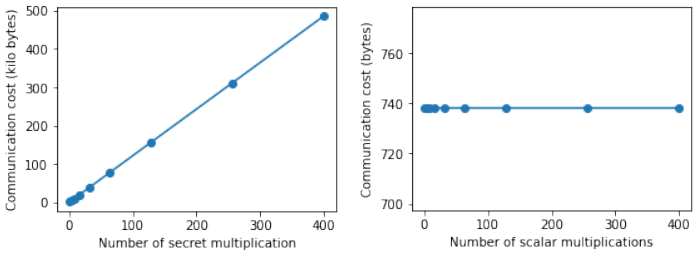

Fig. 3: *Communication cost evolution with nb of secret/scalar multiplications*

2) Running time evaluation: For the running of time evaluation, we created test cases for all desired settings and parameters with a function to generate the *parties* and *expression* variables easily. Each experiment used a running server in the background and we benchmarked the run time of: the creation of client processes, their startup, until the last client's process joined. This can be seen as the maximum run time over all clients (we consider the time to traverse the client list as negligible, the for loop being capped to 64 clients at most). This metric is the most meaningful in our opinion for this protocol: every client is expected to require the final share of every other client, meaning that in a non-active adversarial model, every client must have the final result available at the same time. The results are computed as averaging the run iterations' run time. We used a minimum of 25 iterations in the hope of having meaningful results with relation to the Central Limit Theorem. The results shown are for the 8cores arm64 M1 Chip on Ubuntu 22.04.

# parties	min	mean	median	max	stddev
2	0.009	0.010	0.010	0.013	0.001
8	0.052	0.059	0.054	0.103	0.012
32	0.631	0.663	0.650	0.734	0.029
64	2.462	2.650	2.640	2.943	0.125

Tab. I: *Running time (in ms) for increasing # of parties on a single addition*

# scal_add	min	mean	median	max	stddev
1	0.012	0.013	0.013	0.016	0.001
8	0.012	0.013	0.013	0.019	0.001
32	0.012	0.014	0.013	0.022	0.002
128	0.012	0.014	0.013	0.019	0.001
512	0.014	0.015	0.014	0.024	0.002

Tab. II: *Running time (in ms) for increasing # of scalar additions*

V. Application

1) Balelouc negotiations: In order to test our SMCompiler, we test a custom application called Balelouc negotiations. Every year, on the FLEP campus, a big student festival under the name of Balelouc is held. The bars on the festival site are held by student associations, such as CLIC. Three parties, the CLIC Bar, the Balelouc committee and a drink wholesaler, are trying to estimate the conditions under which they could partake in a profitable collaboration together. The CLIC Bar will be in charge of selling drinks during the festival, they would like to keep the number of drinks they estimate they will sell, n_drinks, secret. The Balelouc committee decides the price at which they would like drinks to be sold during their festival, $resale\_price$, and would also like to keep this a secret. The drink wholesaler decides at which price they will sell their drinks to the festival for resale, $base\_price$. Secrecy of these values aims to avoid influencing the negotiations. They would like to obtain an estimation of how much profit they will make, given their secrets. In addition to the secrets, the CLIC Bar agreed to give two free drinks to the FLEP President Vartin Metterli and the three parties decide to compute this profit over 10 years to have a long term vision. Therefore, the overall profit P is computed as follows: $P = (n\_drinks - 2) * (resale\_price - base\_price) * 10$

# scal_mul	min	mean	median	max	stddev
1	0.012	0.013	0.012	0.015	0.001
8	0.012	0.016	0.013	0.056	0.009
32	0.012	0.013	0.012	0.022	0.002
128	0.012	0.014	0.013	0.023	0.002
400	0.012	0.014	0.013	0.022	0.002

Tab. III: *Running time (in ms) for increasing # of scalar multiplications*

# sec_add	min	mean	median	max	stddev
1	0.013	0.014	0.014	0.021	0.002
8	0.013	0.015	0.014	0.022	0.002
32	0.013	0.014	0.013	0.022	0.002
128	0.013	0.015	0.015	0.024	0.002
512	0.014	0.015	0.015	0.024	0.002

Tab. IV: *Running time (in ms) for increasing # of secret additions*

# sec_mul	min	mean	median	max	stddev
1	0.019	0.020	0.019	0.024	0.001
8	0.060	0.066	0.063	0.113	0.011
32	0.200	0.211	0.204	0.273	0.016
128	0.758	0.802	0.790	0.891	0.035
400	2.454	2.678	2.696	2.813	0.101

Tab. V: *Running time (in ms) for increasing # of secret multiplications*

2) Discussion and Countermeasures: The threat is therefore that one party learns the secret of another and modifies it's own secret. For example, if the drink wholesaler initially planned to set the $base\_price$ to 5 but discovers the secret $resale\_price = 15$, the drink wholesaler can comfortably increase its $base\_price$ to 10 in order to maintain a high overall profit while increasing its personal revenue. There is a potential privacy leakage in our custom application. Notably, if the overall profit P has a negative value, either the $base\_price$ is higher than the $resale\_price$, or n_drinks is smaller than two. Therefore, if the latter is false, the CLIC Bar knows that the primer is true, which reveals information on the other parties secrets. In order to mitigate this, we could implement a min circuit which returns $min(0, P)$. This way, we avoid negative values. In this case, if the final value is zero, no party can distinguish between $base\_price = resale\_price$ and $base\_price > resale\_price$. Additionally, parties other than the CLIC bar can not distinguish between the previous cases and the case in which: $n\_drinks \leq 2$. While this mitigation is not perfect, it avoids the CLIC bar from being able to conclude that either other party is significantly over/underestimating its prices.