Analysis of (V)OLE protocols and their usage to construct 2-party digital signature based on Schnorr

Yelyzaveta Kukovska

August 2022

Abstract

In this report, I try to give out my understanding of the new protocol and its applications. Also try to find opportunities to include learned functionality into a remade version of Schnorr and DSA digital signature schemes for multi-party computation. The challenge in creating a new way of two-party Schnorr is in providing a secure path of exchanging the elements of the Signature between a Client and a Server before it can be checked by a Verifier. (Vector) Oblivious Linear Evaluations protocols, which were brought into a cryptography conversations mainly by a group of professors from Danish University of Aarhus and Israel Institute of Technology Technion, could become a cheaper way of parties halves commitment. Before trying to add (V)OLE there is a need to look for a formula of a shared element, which will be similar to a linear formula, from that the necessity of (V)OLE inclusion will be visible and accurate. It could be in KeyGen( ) phase, it could be in Sign( ) or in both. Probably, something would need to be change in order to fit (V)OLE. Then a question will arise in possibility of doing certain alterations to a classic Schnorr scheme algorithm – so the created two-party version of Schnorr with (V)OLE inclusion satisfy the processing standards.

1 (V)OLE protocols

1.1 Concept

One party has parameters for a linear function a, b and other party wants to evaluate this linear function, but we don't want to let the latter party find the coefficients of the function. And for the one party with coefficients we don't want it to know using what x do we count the result of the function, which is y. There are many kinds of Oblivious Linear Evaluations. From Random-OLE to Vector-OLE. So to use OLE functionality we can have all of the inputs, which are x, a and b, sampled at random and still receive a, x, y and b as outputs without private inputs. Or one of the parties can input a vector of elements a and b so at the end we'll receive vectored function applied to the input x: $x \times \overrightarrow{a} = \overrightarrow{y} - \overrightarrow{b}$ becomes $\overrightarrow{y} = \overrightarrow{a} \times x + \overrightarrow{b}$. VOLE we can also define a random variant where the vectors and the input x are sampled by the functionality.

A core primitive, Oblivious Linear Evaluation is becoming a fundamental building block in many secure computation protocols. Its functionality allows a Receiver to learn a secret linear combination of a pair of field elements held by a Sender. It allows a Sender, holding an affine function $f(x) = a + bx$ over a finite field or ring, to let a receiver learn $f(w)$ for a w of the receiver's choice. In terms of security, the Sender remains oblivious of the receiver's input w, whereas the receiver learns nothing beyond $f(w)$ about f. In recent years, OLE has emerged as an essential building block to construct efficient, reusable and maliciously-secure two-party computation. [1] In difference to OLE, usage of Vector-OLE can inputs and outputs n-elements in oneself, which is useful in many cases and is the reason it's been studied a lot recently. VOLE – the arithmetic analogue of string Oblivious Transfer with a two-party functionality.

At the beginning I was looking at VOLE as an continuation of an OLE functionality, which dealt with n in n instances by building a single itself of length n. In many protocols, building VOLE is a little bit easier than OLE, so it's a more popular primitive. VOLE is a core building block of secure computation, which

can be defined over any finite field, ring or any kind of general finite group. The goal in OLE is usually to have Alice and Bob exchange a set of messages at the end of which Alice will output this value y, a Linear Function Evaluated on input x, where the Linear Function is just a pair of elements a and b input by Bob. In VOLE behind those two elements are just more elements in hiding. Meaning what in many schemes we could replace a large number of instances of OLE by a smaller number of long instances of VOLE. This motivates the goal of amortizing the cost of generating long instances of VOLE. Some protocols are taking one element at a time to encrypt, like Pailler Encryption Scheme does.

1.2 Usage

Oblivious Pseudorandom Functions. Actually there's a very natural construction, where you can take a VOLE of some length n and get a batch of Oblivious-PRF evaluations on n different inputs.

We can construct VOLE "non-silently", meaning the process will be done in online phase. In construction of **Linearly Homomorphic Encryption** VOLE looks like that: *Alice, the Receiver* inputs x, a scalar in Z_p. Generates the keys: $pk, sk \leftarrow Gen(1^\lambda)$ and outputs $pk, [x]$ (where [x] is encrypted to be a ciphertext by a secret key). When the scheme operates on vectors then Alice will pack x into every component of the vector in a same way. Meanwhile *Bob, the Sender* , inputs $\vec{a}, \vec{b} \in Z_p^m$. By combining all these elements, Bob sends to Alice $[\vec{y}] = \vec{a} * [x] + [\vec{b}]$. Which then turns into $\vec{y} = Dec_{sk}([\vec{y}])$. In difference to OLE from **Oblivious Transfer**, to make (V)OLE from OT we need to repeat further described algorithm n-times, where n is the length of the vector ($y_i = b_i + ax_i \Rightarrow y = b + ax$): Alice, the Receiver inputs x, a scalar in Z_q and bit-decompose it into x_i bits ($x = \Sigma_{i=1}^{m} 2^{i-1} x_i$). Meanwhile *Bob, the Sender* , inputs $a, b \in Z_q$. Instead of bit-decomposing, he will sample random field elements $b_i \in \mathbf{Z}_q$ s.t. $b = \Sigma_i 2^{i-1} b_i \bmod q$. Need to remember here that if b (often thing in OLE usage) is random then all these b_is are sampled as random Z_q elements. Then the parties will run a batch of Oblivious Transfers, which we repeat log_q times for each of the bits of x , where Bob always input two messages b_i and $b_i + a$ for the same a each time. Alice will output $y = \Sigma_i 2^{i-1} y_i$. Each y_i is equal to either b_i or $b_i + a$ for same a each time. Although both are non-silent protocols, Homomorphic Encryption has different tradeoffs to the OT based approach as the main goal here is to reduce the amount of communication and avoid this quadratic overhead.

Other uses for OLE and VOLE in MPC protocols can be, firstly, more generalized form of multiplication triples: for instance, if you use VOLE instead of OLE to get scalar vector multiplication very naturally. We can build this up in batches to perform matrix vector multiplication or matrix triples. Which helps in building authenticated secret sharings of correlated randomness for the Preprocessing phase; also to generate information-theoretic MACs. VOLE on commitment level. Many use it for commit and prove protocols as VOLE is good in using information-theoretic MACs to achieve high efficiency for boolean and arithmetic circuits with a focus on supporting large circuits. As stated in [1] that's an interactive zero-knowledge proof system, which can adapt to the ring setting. Mac-n-Cheese protocol, which uses VOLE and MACs, can also be non-interactive (after a Preprocessing phase) by using the Fiat-Shamir transform, and with only a small degradation in soundness. So we need to verify multiplications. There are some products are x,y and z. In the simplier version of Mac'n'Cheese the Prover will first commit to some auxiliary commitment c, which is supposed to be $a \times y$ for some random commitment a , which is also taken from a random VOLE. Then the Verifier sends a random challenge, which is just a random field element, to which the Prover response by opening $e \times x - a$ and then the parties together run this insert zero statement to check that $e \times z - c - d \times y$ is zero.

Meanwhile in "silent" preprocessing all the process done before online phase starts, like setting up functionality and local expansion of short seeds to correlated pseudorandomness. Correlated OT or Δ-OT, in the original article of the presentation [2] stated that correlated OT is inspired by the OT extension protocol of Yuval Ishai and it is a restricted form of oblivious transfer given by subfield VOLE . It has useful applications such as garbled circuits and secure computation with information-theoretic MACs.

2 Schnorr digital signature scheme

Setup() :

1. Generate a prime number q
2. Choose $1 < g < q$ such it generates a multiplicative group G of order q
3. Choose hash function $H(x) : \{0,1\}^* \to G$

KeyGen() :

1. $x \leftarrow G$ (sample x from G uniformly at random)
2. $y = g^x \mod q$
3. return $sk = x$ (secret key), $pk = y$ (public key)

Sign(sk, m) :

1. sample a session key $k \leftarrow G$
2. $r = g^k \mod q$
3. hash the message as $e = H(m||r)$
4. calculate signature $s = k - x \cdot e \mod q$
5. Output (s, e)

Verify(pk, m, σ) :

1. if $e = H(m||(g^s pk^e \mod q))$, return 1 (success)
2. Otherwise, return 0 (fail)

Correctness: $g^s pk^e = g^s g^{xe} = g^{k-xe} g^{xe} = g^{k-xe+xe} = g^k = r$

2.1 Analyzing literature on Two-party Schnorr

The history of exploring the possibilities of creating Two-Party computation Schnorr Signature Scheme goes from updating key-halves through a secure channel to including more hash functions. People already were thinking of making Schnorr into secure Multiparty Computation:

1. In an article from New York University Department of Computer Science [3] the authors used study proactive two-party signature schemes in the context of user authentication. The authors made the first, as they said, method of jointly producing signatures and periodically refreshing the sharing process of the secret key. All that to efficiently transform Schnorr popular signature scheme into MPC (as well as Guillou and Quisquater), by dividing r : $r = r_c + r_s$, and S : $S = S_c + S_s$. They assumed to have a secure channel between Server and Client, which they used to update the key halves (x_c and x_s) with randomizing element $\delta \in Z_q$, the result of which ($x'_c = x_c - \delta \mod q$ and $x'_s = x_s - \delta \mod q$) sent via that same channel. Perhaps the use of (V)OLE functionality will illuminate the need of having "one honest key update".
2. Another process of making a Two-Party version is ending with creation of an additional hash function. I stumble upon it in the post by M. Hamilis at HackMD [4], where MuSig algorithm was also described.

3. Multi-Signature in its traditional meaning is a protocol, which allows couple or more parties of signers to produce a joint multi-signature on a common message.

   In 2006 [5] researchers were battling a practical obstacle to multi-signatures by removing key-setup requirements by presenting a multi-signature scheme in the plain public-key model. There is no dedicated key generation protocol. A signer is not assumed to have proved knowledge of its secret key to the CA, but only to have a standard certificate. Yet, security against rogue-key attacks is proved without the Knowledge of Secret Keys (KOSK) assumption.

   but if go deeper [6] MuSig2 two-round multi-signature scheme

4. The research on similar to Schnorr Digital Signature scheme DSA shows how the usage of MtA is a popular way of morphing that scheme into multiparty computation. It is discussed in the 5th chapter of "Fast Multiparty Threshold ECDSA with Fast Trustless Setup"[7] and in many other works as well, there authors using the same technique of Multiplicative to Additive transformation schemes. That is interesting and needs to be tried out in our research.

The only thing, which always stays the same in all the different tries to make Schnorr into MPC – is a Setup( ) phase. From where some core elements are being randomly generated before any signatures schemes have been chosen:

Setup( ) phase

Gen prime q

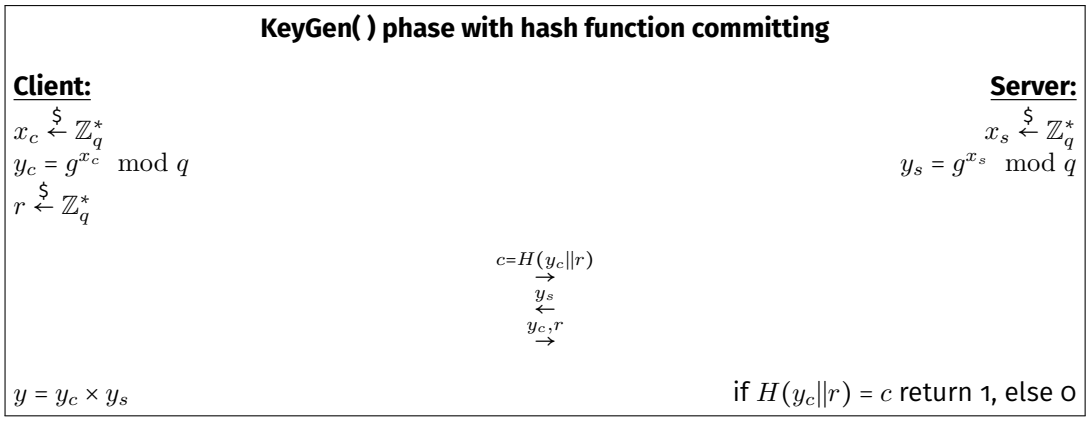
Choose$1 < g < q$

$H(x) : \{0,1\}^* \to G$

After reading through the literature, the main question for me was – is the MPC version of Schnorr can still go through the Verifier check if we add new or loose some of the steps from the classic version?

2.2 KeyGen( ) for Two-party Schnorr in this report

In coming up with the look of Two-Party Computation the most important part is to look for all the sharing elements, which are creating between Client and Server. They are evident from the list of elements for which the Verify( ) phase asks for. In Schnorr's case they are – public key y, e, $Sign$. And the shared element e (if it's going to stay in its original form as a hash function of the original message and element r, which is a g to the power of a session key k mod q) cannot be obtained by adding its two parts from Server and Client, because hash functions are not linear. So, in order to have a shared e, it is necessary to immediately make it so, for which we need a shared r. There are many ways of committing halves to create a combine element, like an RSA algorithm, for example. Where $P_{kenc} : (e, n)$ with $n = p \times q$ and $2 < e < (p-1)(q-1)$. But that specific process of combing two halves is not secure as the Client could send $y_c \times \frac{1}{y_s}$ as y_c , which would eliminate y_s out of the signature latter, and nobody will know until it would be too late. That problem has a name – a Rogue Key Attack – where one of two parties sending , along side its own public key, an inverse public key of the other party, which output came first, to corrupt the shared y. Which would lead to forgery. By trying to prevent this king of attack I ended up using hash function committing for creating a shared public key.



2.3 (V)OLE in Schnorr digital signature for two-party computations

2.3.1 Tries and errors

My first attempt was trying to include VOLE from Linear Homomorphic Encryption into KeyGen(). Initial problem with that was the vector part as, after completion of evaluations and decryption, the shared public key was becoming also $\overrightarrow{y}$ a vector. Is there a possibility to just hash the elements of $\overrightarrow{y}$ to create a simple y without n-elements inside? It would gave up a big amount of bits, which would make our whole scheme work slower. The message will be longer. The saying *"the strength of encryption is related to the difficulty of discovery the key, which in turn depends on both the cipher used and the length of the key"* probably isn't enough of the reasoning to use VOLE instead of OLE. However, even if there will be a switch to OLE the problem with inserting this particular method into our public key still stays. A random b element is left hanging like an unnecessary tail even after decryption: $y = y_c \times y_s + b = g_c^x \times g_s^x + b = g^{x_c + x_s} + b = g^x + b$. I hoped that I came up with the way to interpret or hide this random b somehow further in a scheme, just to keep some version of OLE functionality in KeyGen(), but that gives us no use. And there is a better place to use OLE – a Sign( ) phase.

While looking for solution of dealing with multiplicative shares I was drawn to Mac'n'Cheese commit and prove protocol [8]. For a brief moment I felt like we can use its simple version with Vector OLE , but then I remembered that Schnorr is not interactive in that way as Zero-knowledge related protocols are. However, the good thing for our research I take from this time looking on ZK commit and prove schemes is a simple parenthesis opener.

2.3.2 The process of including OLE into Sign() phase

Schnorr can be seen as a way to prove that we know the private key x by computing a linear function with it: $f(e) = k - xe$. Which is quite similar to OLE, where we essentially prove that we know the coefficient a of the linear function $f(x) = ax - b$.
In Schnorr session key k and hash function element e are related in a certain way, it is not obvious how such a dependence is transferred to OLE. For the case of a distributed secret, the natural desire is to divide a linear function into a sum of several such functions by dividing all the coefficients. It is generally interesting to insert it into the signature itself.

The formula for Schnorr's signature is $Sign = k - x \times e$. While the output from OLE functionality looks like: $y = a \times x + b$. The full look of our $Sign$ in the version of Two-Party Schnorr currently looks like: $Sign = (k_c + k_s) - (x_c + x_s) \times e = (k_c - x_c \times e) + (k_s - x_s \times e)$. So the idea is to brake the combined $Sign$ formula into two parts for Client and Server in such a way that it made them use OLE functionality. Similar thought was presented on the slides of Peter Scholl (V)OLE presentation, where he talks about multiplication triples from OLE [9].

Let's try to visualized the collecting of $Sign$ shares with OLE. The plan is to repeat the functionality two

times for Client to securely received needed evaluation as well as for Server. Starting with Client, who is now supposed to send one element, which is x in OLE protocol model, and in our scheme this is e_c. In the same time Server need to send an a and a random element b, which are k_s and x_s respectively. But if the same element e will be in both parties, then the computation will not be secure. That's a huge problem as we need a combined e for it to be send to Verifier. And the halves cannot be added because of the creation of e from the hash function, which is not linear, meaning $H(a) + H(b) \neq H(a + b)$. For that exact reason the formula for e in the Sign( ) phase has been changed from the original Schnorr's $e = H(m||r)$ to $e_c = r_c \times H(m)$ and $e_s = r_s$. The decision to put $H(m)$ only in one share of e was made so there will be no clutter in the Verifier( ) phase. However this change alone is still not enough to make e additive, because of its elements r, which have a power of secret keys x_c and x_s to randomly chosen g. So the addition will simply not be supported by the rules of math. Something else needs to be done. Maybe some conversion algorithm has to happen before to make current multiplicative shares of e into additive ones.

Need to reshape $e = e_c \times e_s$ into $e = e_c + e_s$. For that challenge the literature on how to transfer multiplicative shares into additive has been searched. The revelation was the fact of how popular this is in creating MPC out of DSA digital signature scheme. For example in the work of G.Tillem and O.Burundukov [10] the MtA, which stands for Multiplicative to Additive, is using additively homomorphic Paillier's cryptosystem so DSA can be transformed into a threshold signature scheme. Same can be found in previously mentioned article about DSA MPC [7] in section number 3. The authors create prime elements out of the original ones.

So here is the interpretation of their algorithms using our data and elements: Let's give our Client an α to his e_c and to Server a β to his e_s. So we would have $\alpha + \beta = x = e_c \times e_s$. Client sends $n_c = y_c \times e_c$ to Server. At the same time Server computes the ciphertext $n_s = e_s \times yn_c + y$ and $y_c(\beta') = y_c(e_c e_s + \beta')$, where β' is chosen at random in Z_q^5. Server sets his share to $\beta = -\beta' \mod q$. He responds to Client by sending n_s. Client decrypts n_s to obtain $\alpha\prime$ and sets $\alpha = \alpha' \mod q$. There are some ZK proofs along the way and if both parties are honest, then the protocol correctly computes α, β such that $\alpha + \beta = x \mod q$. And now we can say that α and β are e'_c and e'_s respectively. So we get $e'_c + e'_s = e_c \times e_s$.

MtA scheme

Client:		**Server:**
$x_c \xleftarrow{\$} \mathbb{Z}_q^*$		
	$\xrightarrow{a_c}$	
		Select $\beta' \leftarrow \mathbb{Z}_N$ $a_s = e \times a_c + E_c(\beta') = E_c(e_c \times e_s + \beta')$ Set additive share $\beta = -\beta' \mod q$
	$\xleftarrow{a_s}$	
Decrypt $\alpha' = dec(a_s)$ $\alpha = \alpha' \mod q$		

Although in outline of our Two-Party Schnorr scheme there will be no need for the whole MtA algorithm to be seen. Same can be said about OLE. These kind of functionalities are usually represented by a "black box". That is helpful for the programming stage of the protocol realization, there is an opportunity to chose different kind of MtA from the "library" of similar protocols. So the technique described above can be totally different to the one in future usage of the same Two-Party Schnorr scheme we are now creating. Now we can rearrange the Signature's formula with additive e-halves like this:

$$Sign = Sign_c + Sign_s = ((k_c - x_c \times (e'_c + e'_s)) + (k_s - x_s \times (e'_c + e'_s))) \mod q$$

$$= (-x_c \times e'_c + (k_c - x_c \times e'_s)) + ((k_s - x_s \times e'_c) - x_s \times e'_s)) \mod q$$

$$= ((-x_c \times e'_c + ((k_s - x_s \times e'_c) - x_s \times e'_s + (k_c - x_c \times e'_s)) \mod q.$$

Before looking on visualization of our Sign( ) phase with additive shares of e, there is an important security

thing done in it, which needed to be explained. After finishing already described steps I noticed that Client's half of the element e is being known to Server before OLE functionality, which makes the latter unsecured and unnecessary. So the two steps of transporting e-halves and OLE black-box switched places.

2.3.3 Scheme with additive shares of e

KeyGen( ) phase

Client:

$x_c \xleftarrow{\$} \mathbb{Z}_q^*$
$y_c = g^{x_c} \mod q$
$r \xleftarrow{\$} \mathbb{Z}_q^*$

Server:

$x_s \xleftarrow{\$} \mathbb{Z}_q^*$
$y_s = g^{x_s} \mod q$

$c=H(y_c||r)$ →

y_s ←

y_c, r →

$y = y_c \times y_s$

If $H(y_c||r) = c$ return 1, else 0

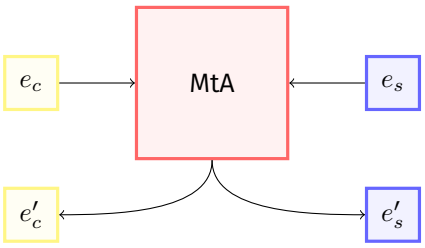
Sign( ) phase

Client:

$k_c \xleftarrow{\$} G$
$r_c = g^{k_c} \mod q$
$e_c = r_c \times H(m)$

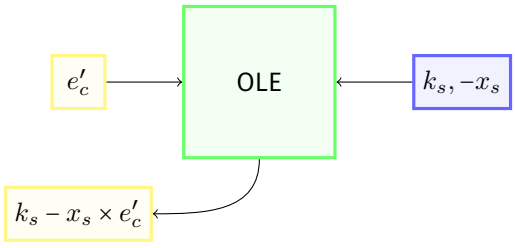
Server:

$k_s \xleftarrow{\$} G$
$r_s = g^{k_s} \mod q$
$e_s = r_s$

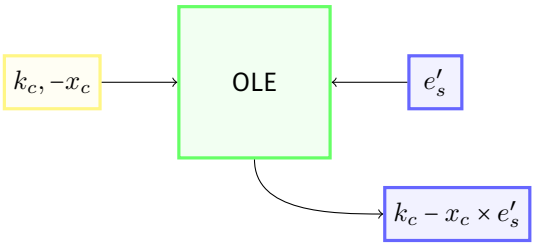


$Sign_c = k_c - x_c \times e$

$Sign_s = k_s - x_s \times e$



$Sign_c = -x_c \times e'_c + (k_s - x_s \times e'_c)$



$Sign_s = (k_c - x_c \times e'_s) - x_c \times e_s$

$\xrightarrow{Sign_c}$

$\xleftarrow{Sign_s}$

$Sign = Sign_c + Sign_s$

$d \xleftarrow{\$} \mathbb{Z}_q^*$

$Sign = Sign_c + Sign_s$

$\xrightarrow{c=H(e_c\|d)}$

$\xleftarrow{e_s}$

$\xrightarrow{e_c,d}$

$e = e_c \times e_s$
$Sign = Sign_c + Sign_s$
If $Sign = Sign'$ return 1, else 0

If $H(e_c\|d) = c$ return 1, else 0
$Sign = Sign_c + Sign_s$
If $Sign = Sign'$ return 1, else 0

Verify( ) phase

If $e = g^{Sign} \times y^e \times H(m)$ return 1, else 0

And the overall output from the Sign( ) phase to Verifier will be e and $Sign$.

After all the work to make additive shares for the combine signature has been successfully done we should take a moment to think why there was a need for such transformation in the first place. Actually there was none. Multiplicative shares of e in Schnorr signature scheme for Two-Parties are working just as fine, which we will soon see in the repeat of the scheme. The parenthesis opener of the combined $Sign$ formula gives us again perfect division to two parts for OLE initiating. It is even easier this time as this scheme wouldn't ask Client or Server to count something locally. The security question behind it was actually the reason why the idea of multiplicative parts was rejected at the beginning, but no reasoning for that was found during the research. So this time the Signature's formula will be rearrange with multiplicative e-halves just like that:

$$Sign = Sign_c + Sign_s = (k_c - x_c \times (e_c \times e_s) + k_s - x_s \times (e_c \times e_s)) \bmod q$$

$$= ((k_c + k_s) - e_c \times e_s(x_c + x_s)) \mod q$$

$$= ((k_s - (e_s \times x_s) \times e_c) + (k_c - (e_c \times x_c) \times e_s)) \mod q$$

2.3.4 Scheme with multiplicative shares of e

KeyGen( ) phase

Client:

$x_c \xleftarrow{\$} \mathbb{Z}_q^*$
$y_c = g^{x_c} \bmod q$
$r \xleftarrow{\$} \mathbb{Z}_q^*$

Server:

$x_s \xleftarrow{\$} \mathbb{Z}_q^*$
$y_s = g^{x_s} \bmod q$

$c = H(y_c \| r)$ →

y_s ←

y_c, r →

$y = y_c \times y_s$

If $H(y_c \| d) = c$ return 1, else 0

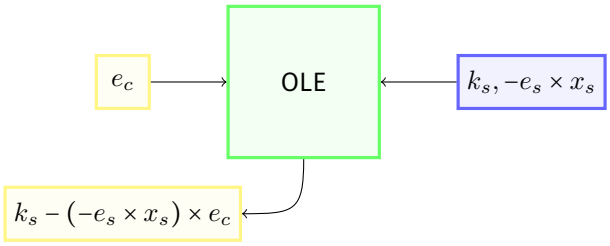
Sign( ) phase

Client:

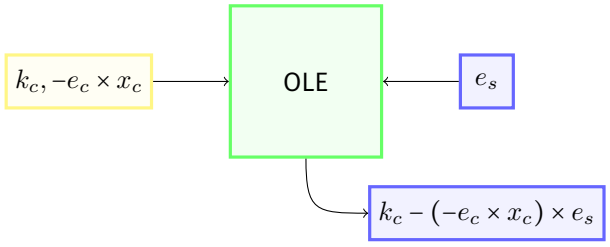
$k_c \xleftarrow{\$} G$
$r_c = g^{k_c} \bmod q$
$e_c = r_c \times H(m)$
$Sign_c = k_c - x_c \times e$

Server:

$k_s \xleftarrow{\$} G$
$r_s = g^{k_s} \bmod q$
$e_s = r_s$
$Sign_s = k_s - x_s \times e$



$Sign'_c = k_s - (e_s \times x_s) \times e_c$



$Sign'_s = k_c - (e_c \times x_c) \times e_s$

$Sign'_c$ →

$Sign'_s$ ←

$Sign' = Sign'_c + Sign'_s$
$d \xleftarrow{\$} \mathbb{Z}_q^*$

$Sign' = Sign'_c + Sign'_s$

$c=H(e_c\|d)$
$\rightarrow$

e_s
$\leftarrow$

e_c, d
$\rightarrow$

$e = e_c \times e_s$
$Sign = Sign_c + Sign_s$
If $Sign = Sign'$ return 1, else 0

If $H(e_c\|r) = c$ return 1, else 0
$Sign = Sign_c + Sign_s$
If $Sign = Sign'$ return 1, else 0

Verify( ) phase

If $e = g^{Sign} \times y^e \times H(m)$ return 1, else 0

Overall there are around 14 communication rounds in the first version with additive shares and 13 in the second with multiplicative e-halves. This amount indicates every time the parties exchanged the message, which is triggered every time the element e is used as its formula contains hashed message.

2.4 Verification phase

Verification check will also change after the re-imagining of earlier phases. Original check is $e = H(m\|(g^s y^e \mod q))$, which is tied to $e = H(m\|r)$ so r is supposed to equal $g^s y^e mod q$ (and let's call it r_v for further notation), if we gonna go with just session key in hash function when it will broke that. And we cannot have k in verification check as it will gave away the private key from the evaluation of the signature: $x = (s - k)/e$.
It is obvious from the observation that if we change the formula for e to $e = r \times H(m)$, when the Verification formula should interpret that also and become something like $e_v = r_v \times H(m)$. Let's check if this new version works with our set of elements:

$g^{k_c} \times H(m) \times g^{k_s}$ should be equal to $g^{((k_c+k_s)-g^{k_c}\times H(m)\times g^{k_s}(x_c+x_s))} \times (g^{x_c} \times g^{x_s})^{g^{k_c}\times H(m)\times g^{k_s}} \times H(m)$:

$$g^{k_c} \times H(m) \times g^{k_s} = g^{((k_c+k_s)-g^{k_c}\times H(m)\times g^{k_s}(x_c+x_s))+(x_c+x_s)\times g^{k_c}\times H(m)\times g^{k_s}} \times H(m) :$$

$$g^{k_c} \times H(m) \times g^{k_s} = g^{(k_c+k_s)} \times H(m)$$

Although the two parts of our new version of Verifier coming together, this still will be too much of a drastic change to the original Schnorr scheme. There is a reason why new cryptography schemes aren't being proposed everyday. They need to follow certain standards. We cannot go against classical Verify check, which meas all this research done of trying to put (V)OLE into Schnorr scheme didn't work out. Was it fruitful in other ways? Answers in conclusion.

2.5 Security check

After already trying to prevent rogue-key attacks by searching for the right building block for our specific exchanges between two parties, there is still a need to prove the security of the whole algorithm. A necessary check is to look out for any linear evaluations with only one unknown. If there is some formula where x_c or k_c by itself send to Server, then it is the only unknown and it can be easily maliciously restored. In the case of formula having two unknowns and Receiver of it can not deduct anything from it. That is just a simple visual check showing that my algorithm makes sense. I did such a review of all the formulas at the end of OLE inclusion process and stumble upon similar problem in the initial final version of the scheme – the parts of e were transferring between the parties before the OLE functionality gets triggered. That left the opportunity to do some sort of rogue attack by corrupting OLE with an inverted e.

However even when no security-weak formulas were found, still it does not mean that the algorithm is safe. It can be dangerous in some non-trivial way. A full security check should be done by reducing the scheme to the discrete logarithm problem. Show that if Server knows how to find Client's private key, or vice versa, having all the entered data, then it knows how to restore x_c, k_c from these numbers - then it knows how to take a discrete logarithm.

3 Next steps

1. Try to come up with an algorithm that, having the invented functionality from this report, is able to calculate discrete logarithms. It may be a trivial challenge or it may not be so simple, but it will definitely help to formally prove safety of the algorithm created here.
2. There is a Fiat-Shamir, which can turn any interactive proof into a non-interactive and still keep all the security.
3. Two-party computation in Schnorr Digital Signature Scheme bares a lot of similarity to Digital Signature Algorithm. There a countless articles and comment sections in cryptography forums about how DSA was created just to go around the patent put on Schnorr and is slightly slower. So the next step will be trying to copy and paste already discussed in this report idea of implementing OLE functionality through rearranging Signature formulas. There we find ourselves a problem of dealing with division. A way to transition $x/k = (x_c + x_s) \times (k_c + k_s)^{(-1)} = (x_c + x_s) \times (k'_c + k'_s)$, where $k'_c + k'_s = 1/k$. So there is a need to research a possible transformation scheme, which could help bring OLE functionality to work in DSA.

4 Conclusion

There is a great need in coming up with new secure ways for transferring the parts of the signature between many parties as the world gets more data and stronger power in malicious decryption. So these new ways should be quick, strong and optimizing. (Vector) Oblivious Linear Evaluations are definitely seems a part. The challenge is to go through every building block and implementation to try to fit it to the one of the standard digital signature schemes. In this report we got through various such methods: from Linear Homomorphic Encryption to Oblivious Transfers. There were some failed attempts of including VOLE into the Key Gen ( ) phase, but that proved to be expensive and unnecessary. So the next experiment was to include OLE into Sign ( ) phase and that become successful after rewriting evaluation of e element a bit and rearranging parts of Signature's formula. Two versions of MPC Schnorr were created with OLE functionality included, as the Multiplicative to Additive transformation gave us the other way of looking into e-halves. The security check also made it necessary to change the order of two steps in that same phase. All these changes done to the original version of the digital signature scheme made it impossible to keep the Verifier check stay the same. It needed to adapt to rewritten e element with different way of hashing the message.

After completing the research, I gained a valuable insight into the importance of maintaining the Verifier check in its original state to ensure adherence to established international standards. Although this protocol may fall short of achieving its intended goal of incorporating (V)OLE into Two-Party Schnorr, it doesn't diminish the value of the work conducted. Throughout the research process, we delved deep into the mechanics of (V)OLE and the Schnorr signature scheme. By attempting to merge a new mechanic with a classical method through multiparty computation, we discovered two potential working versions for the Sign() phase. Should the standard check be altered in the future, these ideas could prove useful. Not all research endeavors yield positive answers or immediate solutions to the presented task. Sometimes, the outcome simply does not work as intended. However, such findings can serve as indicators for necessary changes in standards, particularly if the underlying security proves its worth.

References

[1] C. Baum, A. J. Malozemoff, M. B. Rosen, and P. Scholl, "Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions." Cryptology ePrint Archive, Paper 2020/1410, 2020. https://eprint.iacr.org/2020/1410.

[2] C. Baum, L. Braun, A. Munch-Hansen, B. Razet, and P. Scholl, "Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and $\mathbb{Z}_{2^k}$." Cryptology ePrint Archive, Paper 2021/750, 2021. https://eprint.iacr.org/2021/750.

[3] A. Nicolosi, M. Krohn, Y. Dodis, and D. Mazie'res, "Proactive two-party signatures for user authentication," https://www.scs.stanford.edu/~dm/home/papers/nicolosi:2schnorr.pdf.

[4] H. MD, "Two-party schnorr," 2021. https://hackmd.io/@matan/schnorr.

[5] M. Bellare and G. Neven, "Multi-signatures in the plain public-key model and a general forking lemma," 2006. https://cseweb.ucsd.edu/~mihir/papers/multisignatures.pdf.

[6] J. Nick, T. Ruffing, and Y. Seurin, "Musig2: Simple two-round schnorr multi-signatures," 2021. https://eprint.iacr.org/2020/1261.pdf.

[7] R. Gennaro1 and S. Goldfeder, "Fast multiparty threshold ecdsa with fast trustless setup," 2019-2021. https://eprint.iacr.org/2019/114.pdf.

[8] P. Scholl, "Scalable zero-knowledge protocols from vector-ole." http://cyber.biu.ac.il/wp-content/uploads/2021/11/Vector_Oblivious_Linear_Evaluation-2.pdf.

[9] P. Scholl, "(vector) oblivious linear evaluation: Basic constructions and applications." http://cyber.biu.ac.il/wp-content/uploads/2021/11/Vector_Oblivious_Linear_Evaluation-1.pdf.

[10] G.Tillem and O.Burundukov, "Threshold signatures using secure multiparty computation," https://www.ingwb.com/binaries/content/assets/insights/themes/distributed-ledger-technology/ing-releases-multiparty-threshold-signing-library-to-improve-customer-security/threshold-signatures-using-secure-multiparty-computation.pdf.