Lecture Notes for Cryptographic Computing
7. Oblivious Transfer (Active Security)

Lecturers: Claudio Orlandi, Peter Scholl, Aarhus University

October 23, 2023

In this note we discuss how to implement OT protocols which are secure against active adversaries.

First we look at a very generic technique to convert any protocol with passive security into a protocol with active security, namely the GMW-compiler. We present how this generic technique works by giving a specific example, namely the passive secure OT protocol from last week (the one based on the "oblivious public key generation algorithm" OGen that produces public keys that are indistinguishable from real public keys). Afterwards, we look at a specific construction of efficient OT with active security based on the decisional Diffie-Helmman assumption (DDH), which is far more practical than the general approach.

1 GMW-Compiler

This is an informal presentation of the GMW-compiler. More details can be found in [Gol04, Section 7.4].

1.1 Tools

We are going to use two tools from CPT: Commitments and Zero-Knowledge Proofs of Knowledge.

Commitments: When a sender S commits to a value x for a receiver R, R should not learn any information about x (hiding property). At the same time, when (if) S decides to reveal the committed value to R, she cannot change her mind (binding property). For the sake of concreteness, here is a commitment protocol based on the discrete logarithm assumption: Let (G, p, g) be a group of prime order p generated by g.

- (Public key generation, run once) R chooses a random h that generates G.
- (Commitment Phase) S wants to commit to a value $x \in \mathbb{Z}_p$. S chooses a random value $r \in \mathbb{Z}_p$, computes $c = g^x h^r$ and sends c to R.
- (Open Phase) S sends R (x', r'). If $c = g^{x'} h^{r'}$ R outputs x' or $\perp$ otherwise.

This commitment scheme is (unconditionally) hiding, because h^r is a uniformly random element in G. The commitment is (computationally) binding assuming that the discrete logarithm problem is hard in G, because given two openings $(x, r), (x', r')$ one can compute the discrete logarithm of h in base g.

Zero-Knowledge: We use ideal ZK-functionalities i.e., a model where parties have access to ideal boxes which on common input x and private input w from one of the parties outputs $R(x, w)$. Such a box can in practice be replaced with any of the ZK protocols for NP-complete languages from CPT.

Exercise 1. (Zero-Knowledge for NP)

Look at your CPT notes and find a zero-knowledge protocol for your favourite NP language.

1.2 The Compiler

Passive Secure OT Protocol. Remember that we can implement passive secure OT using any PKE scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{OGen}, \mathsf{OGen}^{-1})$ where one can generate public keys using an oblivious generation algorithm: Alice sends (pk_0, pk_1) to Bob where pk_b is computed using Gen and pk_{1-b} is computed using OGen – the assumption is that fake and real public keys are indistinguishable, so intuitively this message "does not leak information" about b.

In the second message Bob replies $e_i = \mathsf{Enc}(pk_i, m_i)$ for $i = 0, 1$ and Alice decrypts e_b and learns m_b. Intuitively, Alice does not know the secret key for pk_{1-b} and therefore she cannot learn m_{1-b} (more formally, the encryption scheme is IND-CPA secure also for obliviously generated keys).[1]

Attack vs. Passive Protocol. The main problem is that an actively corrupted Alice can sample both public keys using Gen and learn both messages. Note that since the output of Gen and OGen are indistinguishable, Bob has no way of detecting that Alice is cheating.

First Attempt (Add ZK proof). In the first attempt at making the protocol secure against a corrupted Alice we use an ideal ZK proof to let Alice prove that she generated at least one of the two public keys using OGen. In other words, after sending (pk_0, pk_1) to Bob, the parties use the ZK box for the following relation: the statement is $x = (pk_0, pk_1)$ and the witness is $w = r$ and the relation R accepts if

$$(pk_0 = \mathsf{OGen}(r) \vee pk_1 = \mathsf{OGen}(r))$$

Bob learns the output of the ZK box and, in case the ZK box outputs 1 Bob continues the protocol or aborts otherwise.

Attack vs. First Attempt. We show that the protocol is not secure yet. The attack is very subtle, and it involves choosing Alice's randomness r in a malicious way: Alice can 1) compute $pk_0 = \mathsf{Gen}(sk_0), pk_1 = \mathsf{Gen}(sk_1)$, 2) set $r = \mathsf{OGen}^{-1}(pk_0)$ and 3) use r to run the ZK proof.[2]

Second Attempt (Coin flip into the well). We have seen that Bob cannot let Alice choose her own randomness. On the other hand, Bob cannot learn Alice's randomness otherwise the protocol would be trivially insecure. To solve this apparent dilemma, we use a tool called *coin-flipping into the well*: The name comes from the following physical analogy: Bob flips a coin for Alice (so she cannot bias it) by throwing the coin in a well placed at Alice's feet (so she can look down) but far from Bob (so he cannot look inside). We implement this using a commitment scheme Com in the following way:[3]

1. Alice chooses a random string r_A and randomness s for the commitment scheme. She computes $c = \mathsf{Com}(r_A, s)$ and sends c to Bob;
2. Bob chooses a random string r_B and sends it to Alice;
3. Alice outputs $r = r_A \oplus r_B$ and s; Bob outputs (c, r_B);

At the end of the protocol r is a uniform random string (since Com is hiding Bob cannot choose r_B as a function of r_A) and Alice cannot change the value (since Com is binding Alice cannot at a later point in time change r_A).

Now the second attempt at the compiled protocol works as follows: Alice and Bob run the "coin-flip into the well" protocol described above, then Alice generates $pk_b = \mathsf{Gen}(sk)$ and $pk_{1-b} = \mathsf{OGen}(r)$ using the r

[1] If you are still confused about $\mathsf{OGen}, \mathsf{OGen}^{-1}$, just think of a scheme where public keys look like uniform random elements, such as the ElGamal encryption key.

[2] Note that we are using OGen^{-1} on a key generated using Gen, but we argued that if $pk = \mathsf{Gen}(sk)$ then $\mathsf{OGen}(\mathsf{OGen}^{-1}(pk)) = pk$. If this was not the case, this would be an easy way of distinguishing the output of Gen and OGen.

[3] The protocol should remind you of "coin flipping over the phone" from CPT – the only difference here is that the commitment is never opened.

defined by the protocol above. Alice sends (pk_0, pk_1) to Bob and finally Alice and Bob use the ZK box for the following relation: the statement is $x = (pk_0, pk_1, c, r_B)$, the witness is $w = (r, s)$ and relation outputs 1 if:

$$c = \mathsf{Com}(r \oplus r_B, s) \wedge (pk_0 = \mathsf{OGen}(r) \vee pk_1 = \mathsf{OGen}(r))$$

In other words, Alice proves to Bob that at least one of the two public keys was computed using OGen, *using the randomness defined by the coin-flip into the well.*

Attack vs. Second Attempt. The "second attempt" is essentially enough for compiling our passive OT protocol, but in general a third step is needed. Before describing the solution, we present a different OT protocol which is secure against passive adversaries but not active corruptions using the compiler described above.[4] Here is the modified passive protocol:

1. Alice generates $pk_b^1 = \mathsf{Gen}(sk^1)$ and $pk_{1-b}^1 = \mathsf{OGen}(r^1)$ and sends (pk_0^1, pk_1^1) to Bob;
2. Bob computes $e_0^1 = \mathsf{Enc}(pk_0^1, m_0), e_1^1 = \mathsf{Enc}(pk_1^1, m_1)$ and sends (e_0^1, e_1^1) to Alice;
3. Alice generates $pk_b^2 = \mathsf{Gen}(sk^2)$ and $pk_{1-b}^2 = \mathsf{OGen}(r^2)$ and sends (pk_0^2, pk_1^2) to Bob;
4. Bob computes $e_0^2 = \mathsf{Enc}(pk_0^2, m_0), e_1^2 = \mathsf{Enc}(pk_1^2, m_1)$ and sends (e_0^2, e_1^2) to Alice;
5. Alice outputs $m_b = \mathsf{Dec}(sk^1, e_b^1)$;

The protocol simply runs two copies of the original protocol one after another *using the same inputs.* It is easy to see that the protocol is still secure against passively corrupted adversaries (since Alice does not learn the secret keys corresponding to pk_{1-b}^1 and pk_{1-b}^2). If we compile the protocol as described in "second attempt" we get:

1. Run the coin-flip into the well protocol: Alice gets (r, s) and parses $r = (r^1, r^2)$, Bob gets (c, r_B);
2. Alice generates $pk_{1-b}^1 = \mathsf{OGen}(r^1)$ and sends (pk_0^1, pk_1^1) to Bob;
3. Alice and Bob run a ZK proof where Alice proves that $pk_0^1 = \mathsf{OGen}(r^1)$ or $pk_1^1 = \mathsf{OGen}(r^1)$;
4. Bob computes and sends (e_0^1, e_1^1) to Alice;
5. Alice generates $pk_{1-b}^2 = \mathsf{OGen}(r^2)$ and sends (pk_0^2, pk_1^2) to Bob;
6. Alice and Bob run a ZK proof where Alice proves that $pk_0^2 = \mathsf{OGen}(r^2)$ or $pk_1^2 = \mathsf{OGen}(r^2)$;
7. Bob computes and sends (e_0^2, e_1^2) to Alice;

This is clearly insecure, since nothing prevents Alice from learning the secret keys for pk_0^1 and for pk_1^2. In other words, a cheating Alice can change her input from step 2 to step 5, and therefore learn both messages.

Third and Final Attempt (Commitment to Input). To complete the GMW-compiler, we ask Alice to commit to her input bit at the beginning of the protocol and to prove that everything she is doing is consistent with *both* her input and the randomness she gets from the coin flip box. To fully achieve the definition of security, we also need Alice to *prove that she knows the opening to the commitments* (so that the simulator can extract her input). The final protocol (compiled against corrupted Alice) follows:

1. Alice with input bit b chooses random r_A, s, t, computes $c = \mathsf{Com}(r_A, s)$ and $d = \mathsf{Com}(b, t)$. Alice sends (c, d) to Bob;

[4] As many other counterexamples you have encountered in your career, this might look very artificial, but there are plenty of "natural" protocols which suffer from the same vulnerability.

2. Alice and Bob engage in a ZK proof where the statement is $x = (c, d)$, the witness is $w = (b, r_A, s, t)$ and $R(x, w) = 1$ only if:
$$c = \mathsf{Com}(r_A, s) \wedge d = \mathsf{Com}(b, t)$$

3. Bob chooses a random r_B and sends it to Alice;

4. Alice defines $r = r_A \oplus r_B$; Alice chooses a random sk and computes $pk_b = \mathsf{Gen}(sk)$ and $pk_{1-b} = \mathsf{OGen}(r)$; Alice sends (pk_0, pk_1) to Bob;

5. Alice and Bob engage in a ZK proof where the statement is $x = (c, d, r_B, pk_0, pk_1)$, the witness is $w = (b, r_A, s, t)$ and $R(x, w) = 1$ only if:
$$c = \mathsf{Com}(r_A, s) \wedge d = \mathsf{Com}(b, t) \wedge pk_{1-b} = \mathsf{OGen}(r_A \oplus r_B)$$

6. Bob sends (e_0, e_1) to Alice where $e_i = \mathsf{Enc}(pk_i, m_i)$;

7. Alice outputs $m_b = \mathsf{Dec}(sk, e_b)$;

We prove that the protocol is secure against an actively corrupted Alice by constructing a simulator. Remember that in the hybrid model the simulator simulates all calls to the ZK box. In other words, every time Alice inputs something to the ZK box the simulator learns w. The simulator uses the corrupted A as a subroutine and works as follows:

1. S receives (c, d) from A;

2. S receives $w = (b, r_A, s, t)$ from A;

3. S sends random r_B to A;

4. S receives (pk_0, pk_1) from A;

5. S receives $w' = (b', r'_A, s', t')$ from A;

6. If $w' \neq w$ or $pk_{1-b} \neq \mathsf{OGen}(r_A \oplus r_B)$ abort. Otherwise the simulator inputs b to the ideal functionality and learns m_b;

7. The simulator computes $e_b = \mathsf{Enc}(pk_b, m_b)$ and $e_{1-b} = \mathsf{Enc}(pk_{1-b}, 0)$ and outputs the view for Alice
$$(b, t, r, s, r_B, e_0, e_1)$$

We informally argue that the view is indistinguishable from the view of a corrupted party in the real protocol. Note that there are only two differences between the real protocol and the simulation:

1. The simulation aborts if $w' \neq w$ while the real protocol continues as long as $\mathsf{Com}(b', t') = \mathsf{Com}(b, t)$ and $\mathsf{Com}(r'_A, s') = \mathsf{Com}(r_A, s)$. But if this happens then we can use A to break the binding property of Com (thus reaching a contradiction);

2. In the simulation $e_{1-b} = \mathsf{Enc}(pk_{1-b}, 0)$ while in the protocol $e_{1-b} = \mathsf{Enc}(pk_{1-b}, m_{1-b})$; If the distinguisher succeeds then we can break the IND-CPA security of Enc in the following way:
   (a) Receive pk from the IND-CPA challenger;
   (b) Query the IND-CPA challenger with messages m_{1-b} and 0 and receive a challenge ciphertext e^*;
   (c) Construct a simulated view where $r_B = r_A \oplus \mathsf{OGen}^{-1}(pk)$ and $e_{1-b} = e^*$ and feed it to the distinguisher: if the distinguisher thinks the view is from a real protocol, guess 0 in the IND-CPA game; if the distinguisher thinks the view is simulated, guess 1 in the IND-CPA game;

Exercise 2.

"Compile" Bob's side of the protocol as well and argue security against a malicious Bob – write both the protocol and the simulator and argue that the joint distributions of the view of corrupted Bob and output of honest Alice are indistinguishable.

Final Considerations. The GMW compiler is a very useful theoretical tool: it allows to take *any* passive secure protocol and turn it into an actively secure one. Moreover, all the tools used can be based on one-way functions only, which are the minimal assumption for cryptography. All you need to construct ZK protocols are commitments, and the next exercise shows that you can securely implement commitments using pseudorandom generators (which in turn can be constructed based on any one-way function).

Exercise 3. (Commitment from Pseudorandom Generators)

This is a beautiful result by Naor who shows how to construct a secure commitment scheme based on a pseudorandom generator $G : \{0,1\}^n \to \{0,1\}^{3n}$. The protocol is the following:

Commit Phase: Alice wants to commit to a bit b.

- Bob send a random string $y \in \{0,1\}^{3n}$ to Alice.
- Alice picks a random seed $s \in \{0,1\}^n$ and computes $c = G(s) \oplus b \cdot y$ (that is, $c = G(s)$ if $b = 0$ and $c = G(s) \oplus y$ otherwise.

Open Phase: Alice sends b, s to Bob, who accepts iff $c = G(s) \oplus b \cdot y$.

Is this commitment hiding? Why? Is this binding? Why?

Unfortunately, in most cases the GMW-compiler leads to very inefficient protocols. In some cases it is possible to use to use ZK proofs that are efficient for special languages (such as the ZK proofs for discrete logarithm-based statements that you have seen in CPT) – this require the protocol designer to choose tools which are well compatible with each other.

2 Efficient Active Secure OT

In this section we look at a specific OT protocol based on the DDH assumption, with security against active attacks. The protocol still follows the general blueprint of generating two public-keys in such a way that Alice only learns the secret key for one of them, but this time to make sure that Alice does not learn both keys even if she is actively corrupted we ask that the two keys must be generated in a correlated way.

2.1 Active OT from ElGamal

We will start by applying the GMW paradigm to the simple OT protocol based on the ElGamal encryption scheme which we have seen in the notes for OT with passive security. We will exploit the specific properties of keys in the ElGamal encryption scheme to greatly simplify the GMW-compiler in this special case, thus resulting in an overall reasonably efficient protocol which does not require to use generic ZK-proofs, but only very efficient ZK-proofs. Then, we will see some further optimizations and look at some ways in which one might argue that the ZK-proofs might in fact not be necessary after all.

The following protocol uses the ideas first introduced in [BM89,NP01]. The protocol follows the blueprint of the simple passive secure protocol i.e., the receiver Alice sends two public keys to the receiver Bob who

encrypts his messages under the two keys. However, since we want active security and we do not trust the receiver Alice not to learn both secret keys, we introduce an additional round in which the sender Bob sends some random challenge Δ to the receiver. Now the receiver Alice needs to generate two public keys whose product is equal to Δ. Intuitively, this implies that the receiver Alice cannot freely choose both keys. Here is the resulting protocol, where (G, p, g) is a group of prime order p generated by g where the DDH problem is hard.

Challange: Bob picks a random $\delta \in \mathbb{Z}_p$ and sends $\Delta = g^\delta$ to Alice;

Choose: Alice (with choice bit b):

1. Samples random $\alpha \leftarrow \mathbb{Z}_p$;
2. Compute $h_b = g^\alpha$;
3. Let $h_{1-b} = \Delta h_b^{-1}$;
4. Send (h_0, h_1) to Bob;

Transfer Phase: Bob (with input messages m_0, m_1):

1. Abort if $h_0 \cdot h_1 \neq \Delta$; Else:
2. Computes $c_0 = \mathsf{Enc}_{h_0}(m_0; r_0)$ and $c_1 = \mathsf{Enc}_{h_1}(m_1; r_1)$ using random r_0, r_1, where Enc is the ElGamal encryption function;
3. Sends (c_0, c_1) to Alice;

Retrieve Phase: Alice outputs $m_b \leftarrow \mathsf{Dec}(sk, c_b)$, where Dec is the ElGamal decryption function;

In some sense, you can relate the challenge phase of the protocol to the "coin-flip" phase of the GMW compiler, since in both cases this helps preventing Alice from freely choosing both keys. However, in the specific case of ElGamal (thanks to the homomorphic properties of exponentiation) this can be done in a very simple way. There are some interesting security properties that this protocol already satisfies:

Exercise 4.

Prove that Alice, even if she behaves dishonestly, cannot learn both the secret keys corresponding to h_0 and h_1;
Hint 1:

Exercise 5.

Prove that Bob, even if he behaves dishonestly, cannot learn b.
Hint 1:

Unfortunately, the properties above are not enough to prove active security as we have defined earlier. As we have seen several times already our definition requires being able to simulate the protocol having only access to the ideal functionality, which implies being able to "extract" some inputs from the corrupt parties that produce the same output distirbution in the ideal world. To achieve this we can enhance the protocol by adding some ZK proofs. The added ZK proofs are highlighted like this.

Challange: Bob picks a random $\delta \in \mathbb{Z}_p$ and sends $\Delta = g^\delta$ to Alice; Bob and Alice engage in a ZK proof where the statement is $x = \Delta$, Bob has witness $w = (\delta)$ and the protocol outputs $R(x, w) = 1$ to Alice only if $\Delta = g^\delta$;

Choose: Alice (with choice bit b):

1. Samples random $\alpha \leftarrow \mathbb{Z}_p$;
2. Compute $h_b = g^\alpha$;
3. Let $h_{1-b} = \Delta h_b^{-1}$;
4. Send (h_0, h_1) to Bob;
5. Alice and Bob engage in a ZK proof where the statement is $x = (h_0, h_1)$, Alice has witness $w = (\alpha)$ and the protocol outputs $R(x, w) = 1$ to Alice only if $h_0 = g^\alpha \vee h_1 = g^\alpha$;

Transfer Phase: Bob (with input messages m_0, m_1):

1. Abort if $h_0 \cdot h_1 \neq \Delta$; Else:
2. Computes $c_0 = \mathsf{Enc}_{h_0}(m_0; r_0)$ and $c_1 = \mathsf{Enc}_{h_1}(m_1; r_1)$ using random r_0, r_1, where Enc is the ElGamal encryption function;
3. Sends (c_0, c_1) to Alice;

Retrieve Phase: Alice outputs $m_b \leftarrow \mathsf{Dec}(sk, c_b)$, where Dec is the ElGamal decryption function;

Note that the ZK proofs that we use in this protocol can be implemented *much more efficiently* than the generic ones used in the GMW-compiler. If you have followed CPT, you know about the simple and efficient Schnorr protocol for proving knowledge of discrete logarithms (Challenge step). Using an OR-proof, the ZK proof for the (Choose) step is essentially equivalent to two DL-proofs.

We now prove security against a corrupted Alice in the above protocol using the following simulator: (Challenge) The simulator S acts as an honest Bob in the challenge step, sends a value Δ to Alice and simulates the ZK proof. (Choose) Then the simulator receives (h_0, h_1) from Alice. Moreover the simulator extracts α from the ZK proof. Call b' the value such that $h_{b'} = g^\alpha$. (Transfer) Then the simulator checks that $\Delta = h_0 \cdot h_1$ like an honest Bob would do and aborts if this is not the case. If the test does not fail, the simulator inputs b' to the ideal functionality and receives $m_{b'}$. Now, the simulator sends to encryptions (c_0, c_1) where $c_{b'}$ is computed like an honest Bob would do with message $m_{b'}$, while $c_{1-b'}$ is computed as an encryption of 0. This concludes the simulation.

By inspection, the only difference between a real protocol and a simulated one is the way the ciphertext $c_{1-b'}$ is computed. For ease of notation assume that $b' = 0$ from now on and therefore the only difference is the encryption c_1 performed under public key h_1.

Assume an adversary can distinguish between the real protocol and the simulated execution. Intuitively this means the adversary is distinguishing the ciphertext c_1. Therefore we can try to use this adversary inside a reduction that breaks the security of the ElGamal encryption scheme. This is not immediate as it might seem. To see why, note that when we reduce the security to the ElGamal encryption, the security experiment of ElGamal will send us a public key, let's call it pk, but that the encryption c_1 that Alice distinguishes inside the OT protocol is performed under h_1 which is generated during the protocol. Luckily, exploiting again the linear properties of the ElGamal keys this is not an issue. At a high-level, the reduction works as follows: the reduction gets pk from the ElGamal security experiment and then sets $\Delta = pk$ for the OT adversary in the (Challenge) step. In the (Choose) step, the reduction receives h_0, h_1 and α s.t., $g^\alpha = h_0$ from the ZK proof. Now the reduction sends $(0, m_1)$ to the ElGamal challenger and receives an encryption $c^* = (c_1^*, c_2^*) = (g^r, m^* pk^r)$ from the ElGamal challenger (where r is of course unknown to the reduction and where m^* is either 0 or m_1). Now the reduction can turn this encryption under pk into an encryption of the same message under h_1 by computing

$$c_1 = (c_1^*, c_2^*) \cdot (1, (c_1^*)^{-\alpha})$$

Doing the math this is equal to

$$c_1 = (g^r, m^* pk^r) \cdot (1, g^{-\alpha \cdot r}) = (g^r, m^* (pk \cdot h_0^{-1})^r) = (g^r, m^* h_1^r)$$

Which is a valid encryption of m^* under key h_1, thus matching the expected view of the OT adversary. Now, the reduction will guess that c^* was an encryption of $m^* = 0$ if the OT adversary says that he is playing against a simulator, and an encryption of $m^* = m_1$ if the OT adversary says that he is playing in the real protocol. The advantage of the reduction in breaking ElGamal is the same as the advantage of the OT adversary in distinguishing between the simulated protocol and the real one.

Exercise 6.

Prove security against an actively corrupt Bob.

2.2 Optimizing the OT Protocol

The previous protocol can be optimized using the following ideas: 1) It is enough to send one value h instead of both h_0, h_1. Instead of letting Bob check that the two keys satisfy the right relation, we can ask Bob to compute the other key from from h, Δ. 2) There is no need to use ElGamal encryption in the transfer phase. Using a principle similar to Diffie-Hellman Key-Exchange, we can let Alice and Bob compute symmetric keys combining the exchanged group elements and exponents, and use a symmetric (shorter and faster) way to encrypt instead. The resulting protocol is as follows, and is essentially the "Simplest OT" protocol as described in [C015]:

Challange: Bob picks a random $\delta \in \mathbb{Z}_p$ and sends $\Delta = g^\delta$ to Alice;

Choose: Alice (with choice bit b):

1. Pick random $\alpha \leftarrow \mathbb{Z}_p$;
2. Compute $h = \Delta^b g^\alpha$, $K = \Delta^\alpha$;
3. Send h to Bob;

Transfer Phase: Bob (with input messages m_0, m_1):

1. Compute $K_0 = h^\delta, K_1 = (h/\Delta)^\delta$
2. Computes $c_0 = \mathsf{Enc}_{K_0}(m_0)$ and $c_1 = \mathsf{Enc}_{K_1}(m_1)$ using a symmetric encryption scheme.
3. Sends (c_0, c_1) to Alice;

Retrieve Phase: Alice outputs $m_b \leftarrow \mathsf{Dec}_K(c_b)$;

Correctness follows from inspection (Note that in the Choose phase Alice generates h in such a way that α knows the discrete logarithm of h if $b = 0$ or the DL of h/Δ if $b = 1$. Thus $K = K_b$.) and has the same security properties as the basic scheme (Alice cannot learn both keys and Bob cannot learn b, even if they misbehave). Note that the protocol here is presented *without* the ZK proofs. This is because this protocol can be shown to be secure *as it is* against malicious adversaries, if one is willing to make some additional assumptions about the interal working of the adversary. More in detail, the protocol is secure if we assume that the adversary only performs operations in the group using the regular group operations and nothing else. In other words, the adversary only knows how to multiply group elements together and how to raise them to exponents he knows, but the adversary cannot perform any other operation on the group elements that exploits their representation. For instance, the adversary cannot compute the XOR of two group elements, since this is not the group operation. This restricted model for the adversary can be justified since for most groups, especially when looking at groups based on (the right kind of) elliptic curves, there

are actually almost no examples of attacks which exploit the structure of the group representation, and the best attacks are "generic" in the sense that they only use the group operations.

This restricted model, typically known as the "generic group model" or the "algebraic group model" has some very useful properties when trying to prove security of cryptographic protocols. For instance, if we assume that the adversary is generic, if we send the adversary some group elements $g_1, \ldots, g_n$ and the adversary outputs some group element h, then it must be the case that $h = \prod_{i=1}^{n} g_i^{x_i}$ for some values $x_1, \ldots, x_n$ known to the adversary. But if the adversary knows these values we can assume that the simulator also can extract these values from the adversary, and we can therefore save all the ZK proofs from the above protocol: now, by definition, when a corrupt Bob sends Δ, then we know that Bob knows Δ's discrete logarithm δ, which we can therefore extract directly from Bob without having to add any ZK proof. This overall intuition was properly studied and formalized in [ABKLX21].

Exercise 7.

Can you come up with an active secure OT protocol based on other encryption schemes? RSA? Pailler? You are allowed to use ZK proofs, but try to only use ZK proofs for simple algebraic statements (like knowledge of discrete logs, etc.)

References

[BM] Mihir Bellare, Silvio Micali
Non-Interactive Oblivious Transfer and Applications
CRYPTO 1989. https://link.springer.com/chapter/10.1007/0-387-34805-0_48

[ABKLX21] Michel Abdalla, Manuel Barbosa, Jonathan Katz, Julian Loss, Jiayu Xu
Algebraic Adversaries in the Universal Composability Framework
ASIACRYPT 2021. https://eprint.iacr.org/2021/1218

[BM89] Mihir Bellare, Silvio Micali
Non-Interactive Oblivious Transfer and Applications
CRYPTO 1989. https://link.springer.com/chapter/10.1007/0-387-34805-0_48

[C015] Tung Chou and Claudio Orlandi
Non-Interactive Oblivious Transfer and Applications
LATICRYPT 2015. https://eprint.iacr.org/2015/267

[HL10] Carmit Hazay, Yehuda Lindell
Efficient Secure Two-Party Protocols
http://lib.myilibrary.com.ez.statsbiblioteket.dk:2048/Open.aspx?id=300348

[Gol04] Oded Goldreich
Foundations of Cryptography. Volume II: Basic Applications
Cambridge University Press, 2004

[NP01] Moni Naor, Benny Pinkas
Efficient Oblivious Transfer Protocols
SODA 2001. http://www.pinkas.net/PAPERS/effot.ps

[PVW08] Chris Peikert, Vinod Vaikuntanathan and Brent Waters
A Framework for Efficient and Composable Oblivious Transfer
CRYPTO 2008. http://eprint.iacr.org/2007/348