Technical Assessment

Carlo Brunetta

July 12, 2024

0.1 Document Structure

Personal thoughts/solving processes are highlighted into grey boxes such as,

> this one! This is where I add mental notes.
> The notes might not be precise or correct but they are part of my solving process.

The document will follow the effective analysis timing.

The problems are iteratively solved following the mantra: *"read, take notes and repeat"*, thus trying to trace a solution shape as quickly as possible. To achieve this, missing steps/problems will be highlighted for future analysis and will be highlighted in blue boxes such as,

> Read the assignments and solve them!

1 Multi-Party Exponentiation

> To solve the problem:
> 1. Read the paper [AAN18] and protocols (both for notation and context of the protocol).
> 2. Point out what must be proven.
> 3. Prove it!
> 4. Check the code and pen-paper design of the proof of concept (PoC) implementation.
> 5. Implement it

1.1 Proving Security

Denote the protocols of interest, Protocol 5 and Protocol 6, as π_5, π_6 of which security is defined in the UC framework. The paper assumes the existence of an ideal *arithmetic black box* functionality $\mathcal{F}_{\mathsf{ABB}}$, defined and instantiated by Damgård et al. [DFK$^+$06]. Assuming the latter is proven secure in the UC framework, to prove π_5, π_6 UC-secure one must show that the exponentiation protocol $\mathsf{exp}(b, [a])$ required in π_5, π_6, not provided by $\mathcal{F}_{\mathsf{ABB}}$, is UC-secure.

The authors report a UC-secure protocol π_2 [DFK$^+$06] of which efficiency depends on the exponent bit-length. They provide a more efficient (independent on input bit-length, dependent on the amount of parties) protocol π_4 which is pointed out to be insecure against active adversaries. This protocol is utilized in π_5 and π_6: briefly, π_4 is executed twice with different inputs, related by an introduced randomness, and a final reconstruction verifies that the protocol computation was indeed correctly executed.

> Check what the suggested active attack on π_4 does and if this can affect π_5 or π_6.

> Composition of non UC-secure protocols (e.g. π_4) is rarely (almost never) UC-secure.
> A quick skim of [DFK$^+$06] shows the inexistence of π_2 as described.
> Maybe the authors [AAN18] considered the exponentiation in the instantiation as part of the functionality protocol?
> Damgård et al. instantiate $\mathcal{F}_{\mathsf{ABB}}$ via Paillier cryptosystem which effectively encrypts by computing several exponentiations **but** this does not represent $\mathcal{F}_{\mathsf{ABB}}$'s operations.

To show UC-security of π_4 (Fig. 1), one should find a simulator $\mathcal{S}$ able to emulate the ideal functionality $\mathcal{F}_{\mathsf{exp}}$ with the execution of π_4, even in the presence of any adversary $\mathcal{A}$. The transcript is later provided to an environment $\mathcal{R}$ which must distinguish between ideal or emulated execution.

1.2 Insecurity of π_4

Observe that the environment $\mathcal{R}$ can instantiate $\mathcal{A}$ with an input Δ not known to the simulator $\mathcal{S}$. This is key to the active attack on π_4, as highlighted in Fig. 1.

```
Protocol $\pi_4$: $\mathsf{exp}(b, [a])$
$\forall_{P_i} : c_i \leftarrow b^{\alpha_i \cdot a_i}$
$[c_i] \leftarrow \mathsf{Share}(c_i)$
$[b^a] \leftarrow \mathsf{Product}\left(\prod_{i=1}^{n} [c_i]\right)$
```

```
Adversarial $P_1 = \mathcal{A}(\Delta)$ execution of $\pi_4$
// Instantiated $\mathcal{A}$ with uniform random $\Delta \neq 1$
$c_1 \leftarrow b^{\alpha_1 \cdot a_1} \cdot \Delta$
$[c_1 \cdot \Delta] \leftarrow \mathsf{Share}(c_1 \cdot \Delta)$
$[b^a \cdot \Delta] \leftarrow \mathsf{Product}\left([c_1 \cdot \Delta] \cdot \prod_{i=2}^{n} [c_i]\right)$
```

Figure 1: Protocol π_4 and $\mathcal{A}$'s malicious execution (highlighted in red) as, w.l.o.g., party P_1.

Allowing the party to independently compute $b^{\alpha_i \cdot a_i}$ opens the adversary to arbitrarily modify its share to any value; more specifically, we consider $b^{\alpha_i \cdot a_i} \cdot \Delta$ where $\Delta \neq 1$ and provided by the environment during the initialization phase. When reconstructing the shares, the output will be $b^a \cdot \Delta$ thus corrupted.

1.3 Insecurity of π_6

Let us report π_6 in Fig. 2 and highlight the attack. $\mathcal{A}$ participates in the exp executions by maliciously injecting the instantiation coefficient $\Delta_1 = \Delta$ and later injects $\Delta_2 = \Delta^{-1}$.

```
Protocol $\pi_6$: $\mathsf{exp}(b, [a])$
$[b^a \Delta_1] \leftarrow \mathsf{exp}(b, [a])$
$[r] \leftarrow \mathsf{sRand}(\mathbb{Z}_p^*)$
$[a'] \leftarrow \mathsf{Product}([a], [r] - [1])$
$\left[b^{a'} \Delta_2\right] \leftarrow \mathsf{exp}(b, [a'])$
$[w] \leftarrow \mathsf{Product}([a], [r])$
$w \leftarrow \mathsf{Open}([w])$
$[r'] \leftarrow \mathsf{sRand}(\mathbb{Z}_q^*)$
$[v] \leftarrow \mathsf{Product}([r'], \mathsf{Product}[\mathsf{Product}([b^a \Delta_1], \left[b^{a'} \Delta_2\right]), b^{-w}] - [1]) + [1]$
$v \leftarrow \mathsf{Open}([v])$
```

Figure 2: Protocol π_6. Highlighted in red the weak exponential executions.

The injected coefficients cancel out and allow the correct opening of v to 1; however the exponentiation shares opens to the incorrect value $b^a \cdot \Delta$. A simulator $\mathcal{S}$ is able to emulate the attack, however it has $\frac{1}{q-1}$ probability of correctly guessing Δ; thus, the environment has a high probability of distinguishing between real execution and ideal functionality. Therefore,

Prop. 1. *Assuming UC-secure $\mathcal{F}_{\mathsf{ABB}}$ exists, consider the protocol π_4 (Fig. 1), then π_6 is UC-insecure.*

From my quick search, π_2 is not defined by Damgård et al. [DFK+06], thus it is not possible to check if it would allow π_6 to be UC-secure.

Future development, search/design a UC-secure instantiation of $\mathsf{exp}(b, [a])$.

1.4 Insecurity of π_5

While writing about the attack on π_6, I noticed that π_5 utilizes the other exponentiation protocol $\mathsf{exp}([b], a)$ (public exponent, private base). This required to check the provided π_7 since $\mathcal{F}_{\mathsf{ABB}}$ does not provide ideal functionality and Damgård et al.'s reported protocol π_1 is not defined in the original paper [DFK+06] (similarly as before).

Regarding security of π_5, observe that the verification code must execute the public exponent exponentiation $\mathsf{exp}([b^a], r)$ of which instantiation is provided by protocols π_1 and π_7, reported in Fig. 3.

Double-check UC-security and definition of Damgård et al.'s in π_1 [DFK+06]

Check if π_7 is secure and, if not, how this can be used to attack π_5

Similarly to before, the adversary injects Δ in the first call of π_5, i.e. $[b^a\Delta_1] \leftarrow \mathsf{exp}(b, [a])$, and later injects Δ^{-1} in the call $\left[r' \cdot \Delta^{-1}\right] \leftarrow \mathsf{exp}(g, [r])$. This attack forces the exponentiation $\mathsf{exp}([b^a\Delta], r)$ to output $[b^{ar}]$ which correctly verifies the computation (i.e. the protocol outputs $v = 1$); however, the shares open to $b^a\Delta$ and, as before, allow the environment to distinguish between real and ideal execution.

```
Protocol $\pi_5$: $\mathsf{exp}(b, [a])$
$[b^a \cdot \Delta] \leftarrow \mathsf{exp}(b, [a])$
$[r] \leftarrow \mathsf{sRand}(\mathbb{Z}_p^*)$
$[a'] \leftarrow \mathsf{Product}([a], [r])$
$\left[b^{a'}\right] \leftarrow \mathsf{exp}(b, [a'])$
$r \leftarrow \mathsf{Open}([r])$
$[r'] \leftarrow \mathsf{sRand}(\mathbb{Z}_q^*)$
$[v] \leftarrow \mathsf{Product}\left[[r'], \mathsf{exp}([b^a \cdot \Delta], r) - \left[b^{a'}\right]\right] + [1]$
$v \leftarrow \mathsf{Open}([v])$
```

```
Protocol $\pi_7$: $\mathsf{exp}([b \cdot \Delta], a)$
$g \leftarrow \mathsf{getGenerator}$
$[r] \leftarrow \mathsf{sRand}(\mathbb{Z}_p^*)$
$[r' \cdot \Delta^{-1}] \leftarrow \mathsf{exp}(g, [r])$
$[c] \leftarrow \mathsf{Product}([r' \cdot \Delta^{-1}], [b \cdot \Delta])$
$c \leftarrow \mathsf{Open}([c])$
$c' \leftarrow c^a$
$[e] \leftarrow -a \cdot [r]$
$[b^a] \leftarrow c' \cdot \mathsf{exp}(g, [e])$
```

Figure 3: Protocol π_5 and π_7. Highlighted in red the attack.

Prop. 2. *Assume UC-secure $\mathcal{F}_{\mathsf{ABB}}$ exists, consider the protocol π_7 [AAN18], then π_5 is UC-insecure.*

There might be an easier attack similar to the π_6's one by doing some injection in the exponent $a + \delta$ but it would require some math to check where to inject Δ^{-1}.

1.5 Proof of Concept Implementation

Before implementing the code, I designed the code pen-and-paper with the goal of highlighting how (Shamir's) secret sharing is effectively incorporated in the protocol.

The code is designed in several interconnected classes, as shown in Fig. 4, which can be described as:

Finite Field implements the finite field operations used by the other classes.

Share represents the secret share from the secret-sharing algorithm.

Point represents the evaluation point obtained from the secret-sharing algorithm and used to identify the party and reconstruct the secret shared.

Party represents the protocol's party with its shares, points and that collaborates in the protocol execution (via $\mathcal{F}_{\mathsf{ABB}}$ and exp). The party has a corruption functionality that stores Δ. The exp function can be set to be used in the attack.

Secret Sharing implements (Shamir's) secret sharing algorithm.

Ideal $\mathcal{F}_{\mathsf{ABB}}$ implements the ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ and has reference to all the parties in the protocol.

$\mathsf{exp}(b, [a])$ implements the protocols $\pi_4, \pi_5, \pi_6, \pi_7$ and a PoC of the attack to π_6.

To run the code, simply run:

```
$ python code/exp/main.py
```

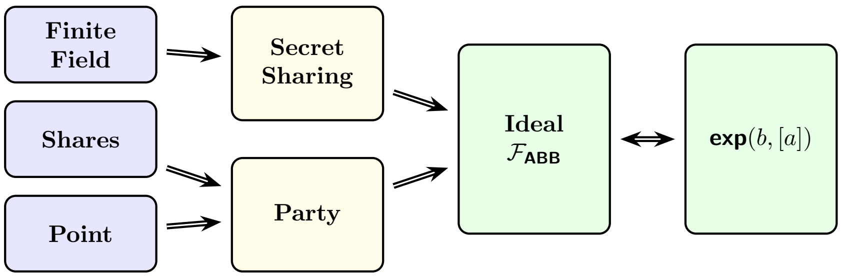

Figure 4: Abstract representation of the implementation subdivided into classes.

Several warnings will appear which relate to specific implementation assumptions:

- There is no input sanitization, i.e. there are no checks nor errors, no handling for wrong parameters n, t, p or the sampled objects Δ, α_i, r_i. Forced *always secure* intervals are used.
- The $\mathcal{F}_{\mathsf{ABB}}$ handles the whole protocol without simulating the effective parties communication.
- π_4, π_7 only work for $n = t$. This is because the parties must compute $b^{\alpha_i \cdot a_i}$ where α_i is the Lagrange coefficient to be used for the reconstruction. This coefficient depends on the parties used in the final reconstruction which modifies the coefficients and the effective protocol execution (which should be limited to the t parties that will reconstruct the final result).
- Shares of exponents and finite field elements live in different groups. The paper itself discusses strategies to do this transformation and points to a specific paper. The PoC code naively opens and shares in the correct space leaving the correct handling for future development.
- The code only contains a showcase of π_6's attack. The attack to π_5 can be similarly implemented.

Further developments:

- clean up the code and better handle the intrinsic difference between shares of exponents and elements of the finite field.
- find/define a secure and efficient protocol for the exponentiation (substitute π_4)
- for completeness, implement attack to π_5

2 Zero Knowledge Proof

Read and report the Σ protocol. Design the implementation and interactions between the classes.

Consider a finite group $(\mathbb{G}, \circ)$ with generator g and order p prime. Observe that any element $x \in \mathbb{G}$ has a unique coefficient $a \in \mathbb{Z}_p$ such that $\underbrace{g \circ \cdots \circ g}_{a \text{ times}} = x$ briefly denoted $\circ^a g$ and that can be used to represent the scalar product of a times of g. Fig. 5 reports Chaum–Pedersen Σ-protocol [S+03] described in a generic finite group $(\mathbb{G}, \circ)$ or order p prime and with g, h generators of $\mathbb{G}$ and $y_1, y_2 \in \mathbb{G}$. The protocol's goal is proving the knowledge of a coefficient x such that $y_1 = \circ^x g$ and $y_2 = \circ^x h$.

```
Prover's Commit()
    $r \leftarrow\!\$\ \mathbb{Z}_p^*$
    $(r_1, r_2) \leftarrow (\circ^r g, \circ^r h)$
    return $(r_1, r_2)$

Verifier's Challenge()
    $c \leftarrow\!\$\ \{0, 1\}$
    return $c$

Prover's Open($r, x, c$)
    $s \leftarrow r - c \cdot x$
    return $s$

Verify($y_1, y_2, r_1, r_2, c, s$)
    $r_1 \overset{?}{=} (\circ^s g) \circ (\circ^c y_1)$
    $r_2 \overset{?}{=} (\circ^s h) \circ (\circ^c y_2)$
    return 1 if checks pass
    return 0 otherwise
```

Figure 5: Chaum–Pedersen Σ-protocol represented as pseudo-code. In red, the prover's secret inputs.

Such an algorithm can be easily transformed into an authentication algorithm by noticing that the protocol instantiation, i.e. the publication of the values $(y_1, y_2) = (g^{x_1}, h^{x_2})$ with $x_1 = x_2$, can be seen as the registration phase and the Σ-protocol as the authentication phase (Fig. 6).

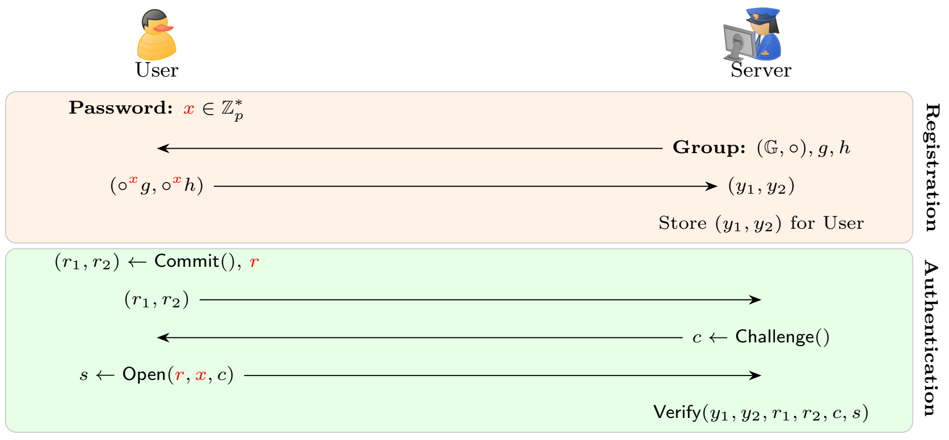
User
Server
Password: $x \in \mathbb{Z}_p^*$
Group: $(\mathbb{G}, \circ), g, h$
$(\circ^x g, \circ^x h)$
(y_1, y_2)
Store (y_1, y_2) for User
Registration
$(r_1, r_2) \leftarrow$ Commit(), r
(r_1, r_2)
$c \leftarrow$ Challenge()
$s \leftarrow$ Open(r, x, c)
Verify$(y_1, y_2, r_1, r_2, c, s)$
Authentication

Figure 6: 1-factor authentication protocol.

2.1 Implementation

The code is designed in several interconnected classes, as shown in Fig. 7, which can be described as:

Finite Group abstracts the group operations required by the Σ-protocol and allows the implementations to the specific instances.

ECC represents an elliptic curve group, instance of finite group.

$\mathbb{Z}_p$ represents a modulo group of order p, instance of finite group.

CP Σ implements Chaum-Pedersen Σ-protocol.

User represents the user functionalities and stores the secret input.

Server represents the server functionalities.

Auth implements the protocol by handling the user and server's communication.

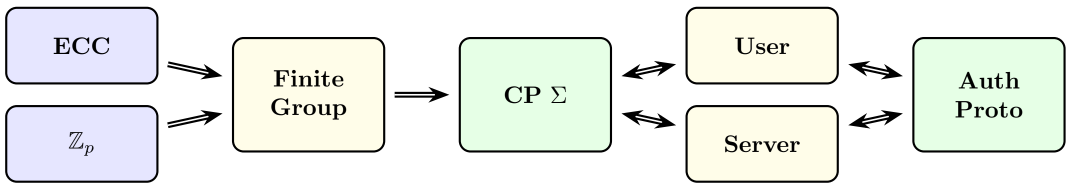

Figure 7: Abstract representation of the authentication proof of concept.

The code will showcase one protocol execution in detail and execute a larger amount of authentication and display the number of failing attempts. To run the code, simply run (the Python package `tqdm` is required for the test progression bar):

```
$ python code/auth/main.py
```

For the sake of the PoC, small groups are chosen and fixed. A note is left when using the implementation to highlight that the code is a PoC. Several limitations are:

- There is no input sanitization, i.e. there are no checks nor errors, no handling for wrong parameters or the sampled objects. Forced *always secure* intervals are used.
- The elliptic curve implementation is general, simplistic and not efficient, i.e. during instantiation the code finds a generator for the biggest prime subgroup in the manifold by brute force and it does not easily support known curves.

Further developments:

- clean up the code and merge with an interface for effective communication
- the elliptic curve code is a simplistic adaptation of the publicly available code
- for real application, usage of side-channel–secure implementations for finite groups is to be preferred and introduced
- for real application, the registration phase is critical and it would improve to ask an immediate authentication to verify the user indeed provided the correct inputs
- for real application, the authentication can be transformed into a non-interactive protocol via Fiat-Shamir transform (or similar)

References

[AAN18] Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova. Practically efficient secure distributed exponentiation without bit-decomposition. In *Financial Cryptography and Data Security*, pages 291–309, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 285–304, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[S$^+$03] Nigel Paul Smart et al. *Cryptography: an introduction*, volume 3. McGraw-Hill New York, 2003.