Elaboration on Code for Provisions and MProvisions

Arijit Dutta, Saravanan Vijayakumaran
Department of Electrical Engineering
Indian Institute of Technology Bombay
arijit.dutta@iitb.ac.in, sarva@ee.iitb.ac.in

I. Provisions

Code for Provisions is available at [1]. We elaborate proof generation and verifications in the following two subsections.

A. Proof Generation

In the Provisions' code base, let us navigate to provisions\src\main\java\edu\stanford\crypto\proof \asstes \AddressProofSystem.java. In implementation, the authors followed additive notation unlike the paper where multiplication notation is followed. As given in the protocol 1 in [2], $u^{(1)}, u^{(2)}, u^{(3)}, u^{(4)}$ are chosen randomly from $\mathbb{Z}_q$ in line 21-24 (the function would be called iteratively, so we can ignore subscript i). `data.Get` gives us g, h, y, b ($b_i = g^{bal(y_i)}$) in line 25-28. Commitments $a^{(1)}, a^{(2)}, a^{(3)}$ are computed accordingly in line 29-31. v (randomness for p), t (randomness for l) are also obtained in line 33-34. s is chosen 1 when the private key is known and p and l are calculated (35-37). Now let us consider the binary proof. This is used to show that s is either 0 or 1. Consider protocol 4 in Provisions [2]. In our case $l = g^x h^y$ is equal to $p = b^s h^v$ (we are ignoring subscript i). `falseChallenge` (line 42) is same as c_f in protocol 4. After proper substitution the following equations hold and calculated accordingly in 40-45.

$$a_0 = h^{u_0} b^{-c_f \cdot s} \tag{1}$$

$$a_1 = h^{u_1} b^{c_f(1-s)}. \tag{2}$$

`challenge` (c in protocol 4) is the hash of (g, h, y, b, p, l,`a1,a2,a3,aZero,aOne`). Responses in protocol 1 are calculated accordingly in line 49-52. The following equations explain the calculation of `responseZero` of protocol 4.

$$r_0 = u_0 + (c - s(c - c_f) - (1 - s)c_f)v \qquad \text{(according to protocol 4 and substitution)} \tag{3}$$

$$= u_0 + (c - c_f)v(1 - s) + c_f vs \qquad \text{(as calculated in line 57).} \tag{4}$$

`responseOne` is similarly calculated from protocol 4 by substituting $c_1 = c_f(1 - s) + (c - c_f)s$ and is given in line 58.

$$r_1 = u_1 + s(c - c_f)v + c_f(1 - s)v. \tag{5}$$

Finally the proof is (p, l, `challengeZero,challengeOne,responseS,responseV,responseT,responseX,responseZero,responseOne`). When the private key is known `challengeZero` $= c_f$ and `challengeOne` $= c - c_f$. When the private key is not known it is the other way around i.e. `challengeZero` $= c - c_f$ and `challengeOne` $= c_f$.

B. Proof Verification

We have seen that the proof contains the extra quantities (p, l), challenges, and responses. In normal interactive protocol, `a1,a2,a3,aZero,aOne` are transmitted by the prover. However, in the present implementation the verifier has to generate these values and then check whether the hash of (g, h, y, b, p, l,`a1,a2,a3,aZero,aOne`) is equal to the "transmitted challenge" or not. This optimization makes the proof size smaller.

Let us now navigate to provisions\src\main $\cdots$ \proof\verification\AddressProofVerifier.java. In line 19, `transmittedChallenge` is calculated as (`challengeZero` + `challengeOne`). Note that in every cases the summation gives c. Variable `balanceClaim` is calculated as p^c and variable `xHatClaim` is calculated as l^c. Now let us see how we can calculate $a^{(1)}, a^{(2)}, a^{(3)}$ in protocol 1 from $p^c, l^c, r_s, r_v, r_t, r_{\hat{x}}$ (we are again ignoring subscript i). First of all, this is to note that $u^{(1)}, u^{(2)}, u^{(3)}, u^{(4)}$ are sampled in order to prove knowledge of the quantities $s, v, t, \hat{x}$ respectively. So we write $u^{(1)}, u^{(2)}, u^{(3)}, u^{(4)}$ as $u^{(s)}, u^{(v)}, u^{(t)}, u^{(\hat{x})}$ respectively. Now $a^{(1)}$ is a commitment to prove the represent $p = b^s \cdot h^v$ (equation 1 in [2]). So,

$$\begin{aligned} a^{(1)} &= b^{u^{(s)}} \cdot h^{u^{(v)}} \\ &= b^{(u^{(s)} + cs - cs)} \cdot h^{(u^{(v)} + cv - cv)} \\ &\overset{(1)}{=} b^{r_s} \cdot h^{r_v} \cdot p^{-c} \qquad \text{(as calculated in line 21).} \end{aligned}$$

Here equality (1) is true because of the fact that $r_s = u^{(s)} + cs, r_v = u^{(v)} + cv$, and $p = b^s \cdot h^v$.

Similarly $a^{(2)}$ is a commitment to prove the represent $l = y^s h^t$ (equation (3) in [2]). So $a^{(2)} = y^{u^{(s)}} h^{u^{(t)}}$ (see protocol 1 of [2]). It is calculated in line 23 as $a^{(2)} = y^{r_s} h^{r_t} l^{-c}$. $a^{(3)}$ is a commitment to prove the represent $l = g^{\hat{x}} h^t$ (equation 5 of [2]). So $a^{(3)} = g^{u^{(\hat{x})}} h^{u^{(t)}}$ (see protocol 1 of [2]). It is calculated in line 24 as $a^{(3)} = g^{r_{\hat{x}}} h^{r_t} l^{-c}$.

Now let us see how `aZero,aOne` of binary ZK proof is calculated (line 25-29). From protocol 4 of [2], they can be calculated from the verification conditions and proper substitutions $(l \to p, g \to b)$.

$$a_0 = h^{r_0} p^{-(c-c_1)} \quad \text{and} \quad a_1 = h^{r_1} (pb^{-1})^{-c_1}. \tag{6}$$

The authors have defined $\texttt{zeroClaim} \coloneqq p^{\texttt{challengeZero}}$ and $\texttt{oneClaim} \coloneqq (pb^{-1})^{\texttt{challengeOne}}$. Finally they calculated `aZero,aOne` as follows.

$$\texttt{aZero} = h^{\texttt{responseZero}} \texttt{zeroClaim}^{-1} \quad \text{and} \quad \texttt{aOne} = h^{\texttt{responseOne}} \texttt{oneClaim}^{-1}. \tag{7}$$

To prove that a_0, a_1 are the same in equations (6) and (7), all we need to show is that for both $s = 0, s = 1$ (i.e. private key is not known and known respectively) cases the following equations hold.

$$\texttt{challengeZero} = c - c_1 \quad \text{and} \quad \texttt{challengeOne} = c_1 \quad (c_1 \text{ is taken from protocol 4 of [2].}) \tag{8}$$

$c_1 = s(c - c_f) + (1 - s)c_f$ is obtained from protocol 4 after substitution. For $s = 0$,

$$\texttt{challengeZero} = c - c_f = c - c_1 \quad \text{and} \quad \texttt{challengeOne} = c_f = c_1.$$

For $s = 1$,

$$\texttt{challengeZero} = c_f = c - (c - c_f) = c - c_1 \quad \text{and} \quad \texttt{challengeOne} = c - c_f = c_1.$$

Hence Condition (8) is proved and we have verified the sanity of the code. Now that we have generated all arguments of the hash, we compute "`computedChallenge`" and verify if it is equal to the `transmittedChallenge` or not. If they are equal, the verifier declares that the proof is correctly generated.

II. MProvisions

Code for MProvisions is written in a similar way that of Provisions. Here the the paper and the code both follow additive notations for group operaions. The code is available at [3].

A. Proof Generation

Let us navigate to the file mprovisions.h in [3]. In the method `GenerateMprovisionsPoa`, p_i (B_i in the paper), m_i(M_i in the paper), and n_i (N_i in the paer) are generated accordingly (line 143-171). Then we have generated commitments $a_i^{(1)}, \cdots, a_i^{(5)}$ ($A_i^{(1)}, \cdots, A_i^{(5)}$ in the paper) using the equations given in (1.b) in the MProvisions protocol. The equations can be explained by similar arguments that is used in the above section to explain about the a_is in Provisions. That is, quantities $u_i^{(1)}, u_i^{(2)}, u_i^{(3)}, u_i^{(4)}, u_i^{(5)}$ are chosen to prove the knowledge of quantities $s_i, k_i, e_i, f_i,$ and $\hat{x}_i$ respectively. And $a_i^{(1)}, \cdots, a_i^{(5)}$ are the initial commitments generated to prove the representations given in equation (20)-(24) in the Section VI-B of the paper.

Now let us look at `aZero` and `aOne`. Those are calculated in the same way that of Provisions'. In protocol 4 of [2], we have $p_i = s_i C_i + k_i G$ in case of $l = g^x h^y$. So the following equations hold.

$$a_{i,0} = u_o G - c_f s_i C_i \tag{9}$$

$$a_{i,1} = u_1 G + c_f (1 - s_i) C_i. \tag{10}$$

c_f (`cF[i]`) is again a randomly generated scalar. `aZero` and `aOne` are calculated accordingly (line 189-198). After we generated all commitments, we calculate the challenge `challenge[i]` as hash of $G, H, pk_i, C_i, p_i, I_i, m_i, n_i$, and all commitments. $r_{k_i}, \cdots, r_{\hat{x}_i}$ are generated according to (1.d) in the MProvisions protocol (line 221-232). For r_{s_i}, we need to consider two cases, i.e. when $s_i = 0$ and $s_i = 1$.

From equation (4) and (5) of the above section and substituting $v \to k_i$, we get expressions for r_0 and r_1 as follows.

$$r_0 = u_0 + (c - c_f) k_i (1 - s) + c_f k_i s \tag{11}$$

$$r_1 = u_1 + s(c - c_f) k_i + c_f (1 - s) k_i. \tag{12}$$

We update `responseZero` and `responseOne` accordingly for cases $s_i = 0, 1$. Like Provisions, when $s_i = 1$, `cZero` (`challengeZero` for Provisions) $= c_f$ and `cOne` (`challengeOne` for Provisions) $= c - c_f$. For $s_i = 0$, it is vice versa. Hence, the proof is generated.

B. Proof Verification

Like Provisions, here also we generate all the commitments (a_is) from the responses and other components, compute the hash likewise the proof generation process, and verify whether the transmitted challenge is equal to the computed challenge. Here also, `transmittedChallenge` = `cZero` + `cOne`. Commitments $a_i^{(1)}, \cdots, a_i^{(5)}$ are calculated as follows (see (1.e) of the MProvisions protocol).

$$a_i^{(1)} = r_{s_i} C_i + r_{k_i} G - c_i p_i \tag{13}$$
$$a_i^{(2)} = r_{s_i} I_i + r_{e_i} H - c_i m_i \tag{14}$$
$$a_i^{(3)} = r_{s_i} pk_i + r_{f_i} H - c_i n_i \tag{15}$$
$$a_i^{(4)} = r_{\hat{x_i}}\ H_p(pk_i) + r_{e_i} H - c_i m_i \tag{16}$$
$$a_i^{(5)} = r_{\hat{x_i}} G + r_{f_i} H - c_i n_i. \tag{17}$$
$$\tag{18}$$

Now, equation (6), (7), and (8) from above section are also valid in case of MProvisions. After proper substitution from Provisions to MProvisions, i.e. $p(l) \rightarrow p_i, h \rightarrow G, b(g) \rightarrow C_i$ we get the following equations.

$$a_0 = r_0 G - c_0 p_i \tag{19}$$
$$a_1 = r_1 G - (p_i - C_i) c_1. \tag{20}$$

`aZeroCal`, `aOneCal` are calculated accordingly. Lastly, the challenge is computed accordingly and its equality with the transmitted challenge is verified for each i.

REFERENCES

[1] Provisions code. [Online]. Available: https://github.com/bbuenz/provisions

[2] G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh, "Provisions: Privacy-preserving proofs of solvency for Bitcoin exchanges," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, New York, NY, USA, 2015, pp. 720–731.

[3] MProvisions simulation code. [Online]. Available: https://github.com/arijitdutta67/MProveAndMProvisions/tree/v0.13.0.4-mprove/tests/mprovisions