Implementing Blind Signatures on Multos Smart Card - Report

Wojciech Wodo, Lucjan Hanzlik

Wrocław University of Science Technology
wojciech.wodo@pwr.edu.pl, lucjan.hanzlik@pwr.edu.pl

1 Introduction

Blind Signatures. David Chaum [1] was the first to propose blind signatures. This cryptographic primitive allows a user to receive a signature from the signer in such a way that the signer should not be able to link a signature to the issuing protocol (*blindness*). To protect the signer we also require *unforgeability*, i.e. without the knowledge of the secret key, one cannot compute a valid signature.

Blind signatures are an interesting research topic as there are still open problems. The goal is to design a two-move blind signatures (also called *round-optimal* [2]) that is efficient and secure under weak assumptions in the plain model, i.e. without random oracles or a setup phase (a common reference string).

Fischlin et al. [3] showed that this is not an easy task. In particular, it is impossible to construct a three-move blind signature scheme with a black-box reduction to a non-interactive problem instance. This results require that the scheme admits a signature derivation checks, i.e. the transcript of communication allows to verify whether the user is able to derive a valid signature in this execution. This leaves room for constructions that bypass this limitations.

Garg et al. [4] were the first to proposed a generic construction that bypasses those impossibility results. However, the solution is not efficient from a practical point of view as it uses fully homomorphic encryption to evaluate a signing circuit. At Eurocrypt'14 Garg and Gupta [5] improved the previous results and proposed the first efficient round-optimal blind signature constructions in the standard model. The authors prove security using a technique called complexity leveraging. However, the computational and communication complexity of the scheme limits its usage in many practical applications.

Fuchsbauer et al. [6] proposed the first practical round-optimal blind signature scheme in the standard model. Their generic construction is based on structure-preserving signatures on equivalence classes (SPS-EQ) which unforgeability implies the unforgeability of the generic construction. On the other hand, blindness is based on an interactive version of the decisional Diffie-Hellman problem. Recently, the authors improved this result [7] and replaced the interactive assumption by a weaker assumption. The disadvantage of both generic constructions is that they cannot be instantiated with all SPS-EQ. What is more, the only instantiation that can be used still requires an interactive assumption [8].

MULTOS. MULTOS is the second most popular operating system for smart cards. The system allows to use multiple applications that are managed by the MULTOS virtual machine. The MULTOS technology is open but overseen by the MULTOS Consortium a body composed of companies which have an interest in the development of the OS and includes smart card and silicon manufacturers, payment card schemes, chip data preparation, card management and personalization system providers, and smart card solution providers.

In comparison to Java Card, which are the most popular smart cards on the market, MULTOS smart cards can be programmed using different programming languages, i.e.: C, Java or MEL. What is more, the MULTOS standard defines a low-level API that allows non-industry developers (e.g. scientists) to create more efficient application, in comparison to the high-level API given by the Java Card API. On the other hand, smart card manufacturer only implement the mandatory functions described by the MULTOS API and some of the optional functions are not supported.

Our Contribution. In this report we focus on the implementation of an unpublished blind signature scheme, which is based on the single-message protocol for the short randomizable signatures presented

by Pointcheval and Sanders [9]. This protocol allows the user to receive a signature under a message committed in a Pedersen commitment. However, the protocol requires the user to proof knowledge of the commitment opening. Common instantiation for this type of proofs require the random oracle model, multiple rounds or the common reference string model.

The implemented blind signature scheme uses the knowledge-of-exponent assumption [10] instead of this proof of knowledge. It is worth noting that knowledge assumptions were already used in other blind signature schemes [11]. However, the construction is based on unknown order groups, thus are fairly inefficient. The scheme also uses a deterministic parameter generator, so the user can 'trust' the group parameters in the signer's public key. The resulting blind signature scheme is not only two-move but works in the plain model, i.e. without random oracle or a common-reference string.

The main contribution of this report is the following. We present details on how to efficiently implement the user side of the scheme on a MultiApp MULTOS smart card provided to us by Gemalto. First we describe in details the implemented blind signature scheme. Then we present the limitations of the MULTOS card used by us and how to improve it to support more operations. Finally, we show how to use this improvements to implement the presented blind signature scheme.

2 Preliminaries

Definition 1 (Elliptic Curves of Prime Order).

Elliptic curve points (with $\mathcal{O}$ defined as point at infinity) with point addition form additive groups. Usually, in cryptography we use elliptic curves over prime fields $\mathbb{F}_p$, for $p > 3$, and defined by the Weierstrass equation $y^2 = x^3 + ax + b \mod p$. By $E(\mathbb{F}_p)$ we denote an elliptic curve over $\mathbb{F}_p$. For two distinct points $P = (x_P, y_P) \in E(\mathbb{F}_p)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_p)$ we define point addition (in affine coordinates) as $R = P + Q = (x_R, y_R)$, where:

$$\begin{aligned} \lambda &= (y_Q - y_P)/(x_Q - x_P) \mod p, \\ x_R &= \lambda^2 - x_P - x_Q \mod p, \\ y_R &= \lambda(x_P - x_R) - y_P \mod p. \end{aligned}$$

Note that for $Q = P$ the value λ is equal to zero. Thus, for the sum $R = P + P$ we use $\lambda = (3x_P^2 + a)/(2y_P) \mod p$. In addition, if $Q = -P = (x_P, -y_P)$, then $P + Q = \mathcal{O}$. It follows that for any point $P \in E(\mathbb{F}_p)$ we have that $P + \mathcal{O} = \mathcal{O} + P = P$.

We now define scalar multiplication for a point $P \in E(\mathbb{F}_p)$ and scalar $k \in \mathbb{Z}_q$ as $[k]P$, where:

$$[k]P = \underbrace{P + \ldots + P}_{k\text{- times}}$$

Definition 2 (Bilinear map). *Let us consider cyclic groups $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$, $(\mathbb{G}_T, \cdot)$ of a prime order q. Let P_1, P_2 be generators of respectively $\mathbb{G}_1$ and $\mathbb{G}_2$. We call $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ a* bilinear map *(pairing) if it is efficiently computable and the following holds:*

Bilinearity: $\forall (S, T) \in \mathbb{G}_1 \times \mathbb{G}_2$, $\forall a, b \in \mathbb{Z}_q$, *we have* $e([a]S, [b]T) = e(S, T)^{a \cdot b}$,
Non-degeneracy: $e(P_1, P_2) \neq 1$ *is a generator of group* $\mathbb{G}_T$,

Depending on the choice of groups we say that map e is of:

Type 1: if $\mathbb{G}_1 = \mathbb{G}_2$,
Type 2: if $\mathbb{G}_1$ and $\mathbb{G}_2$ are distinct groups and there exists an efficiently computable isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$,
Type 3: if $\mathbb{G}_1$ and $\mathbb{G}_2$ are distinct groups and no efficiently computable isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ is known.

Definition 3 (Bilinear-group generator). *A bilinear-group generator is a polynomial-time algorithm* BGGen *that on input of a security parameter λ returns a bilinear group $\mathsf{BG} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, P_1, P_2)$ such that $\mathbb{G}_1 = \langle P_1 \rangle$, $\mathbb{G}_2 = \langle P_2 \rangle$ and $\mathbb{G}_T$ are groups of order q with $\log_2 q \approx \lambda$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map. Similar to [6] we assume that* BGGen *is deterministic (which is the case for BN-curves [12]).*

2.1 Short Randomizable Signatures

We now recall the short randomizable signatures presented by Pointcheval and Sanders [9]. This signature scheme uses type 3 pairing, thus we will use describe it using the bilinear group generator BGGen. We present the modified variant of the scheme that admits the signing of committed messages.

Definition 4 (Short Randomizable Signatures). *The signature scheme is given by the following triple of algorithms given an output* BG *of* $\mathsf{BGGen}(\lambda)$.

$\mathsf{KeyGen}_{\mathsf{CL}}(\mathsf{BG})$**:**
Choose $(x, y) \xleftarrow{\$} (\mathbb{Z}_q^*)^2$*, compute* $X_1 = [x]P_1$, $X_2 = [x]P_2$, $Y_1 = [y]P_1$, $Y_2 = [y]P_2$*, set the public key* $\mathsf{pk}_{\mathsf{CL}} = (\mathsf{BG}, X_2, Y_1, Y_2)$ *and the private key* $\mathsf{sk}_{\mathsf{CL}} = (\mathsf{pk}_{\mathsf{CL}}, X_1)$.

$\mathsf{Sign}_{\mathsf{CL}}(m, \mathsf{sk}_{\mathsf{CL}})$**:**
Select $u \xleftarrow{\$} \mathbb{Z}_q$ *and compute* $\sigma_1 = [u]P_1$, $\sigma_2 = [u](X_1 + [m]Y_1)$. *Output* (σ_1, σ_2).

$\mathsf{Verify}_{\mathsf{CL}}(m, (\sigma_1, \sigma_2), \mathsf{pk}_{\mathsf{CL}})$**:**
Output 1 *if and only if* $\sigma_1 \neq 1_{\mathbb{G}_1}$ *and* $e(\sigma_1, X_2 + [m]Y_2) = e(\sigma_2, P_2)$.

3 The Implemented Blind Signature Scheme

In this section we present the implemented blind signature scheme. The construction uses in fact the single-message protocol presented by Pointcheval and Sanders, for their randomizable signature scheme [9], with some minor changes. The description of the scheme is given is Scheme 1.

$\mathsf{KeyGen}_{\mathsf{BS}}(1^\lambda)$**:** Generate bilinear group parameters $\mathsf{BG} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, P_1, P_2) \leftarrow \mathsf{BGGen}(1^\lambda)$. Compute the short randomizable signature scheme key pair $(\mathsf{sk}_{\mathsf{CL}}, \mathsf{pk}_{\mathsf{CL}}) \xleftarrow{\$} \mathsf{KeyGen}_{\mathsf{CL}}(\mathsf{BG})$, where $\mathsf{pk}_{\mathsf{CL}} = (\lambda, X_2, Y_1, Y_2)$. Compute random $k \xleftarrow{\$} \mathbb{Z}_q$ and set the secret key $\mathsf{sk}_{\mathsf{BS}} = (\mathsf{sk}_{\mathsf{CL}}, k)$. Compute $\hat{P}_1 = [k]P_1$, $\hat{Y}_1 = [k]Y_1$ and the public key $\mathsf{pk}_{\mathsf{BS}} = (\mathsf{pk}_{\mathsf{CL}}, \hat{P}_1, \hat{Y}_1)$.

$\mathsf{Request}_{\mathsf{BS}}(m, \mathsf{pk}_{\mathsf{BS}})$**:** Parse $\mathsf{pk}_{\mathsf{BS}}$ as $((\lambda, X_2, Y_1, Y_2), \hat{P}_1, \hat{Y}_1)$, generate the parameters $\mathsf{BG} \leftarrow \mathsf{BGGen}(1^\lambda)$, choose $t \xleftarrow{\$} \mathbb{Z}_q$ and compute $\rho = ([t]P_1 + [m]Y_1, [t]\hat{P}_1 + [m]\hat{Y}_1)$. Set $\mathsf{St}_{\mathsf{BS}} = (m, t)$ and send ρ to the signer.

$\mathsf{Issue}_{\mathsf{BS}}(\rho, \mathsf{sk}_{\mathsf{BS}})$**:** Parse $\mathsf{sk}_{\mathsf{BS}}$ as (X, k) and ρ as (C_1, C_2). Abort if $[k]C_1 \neq C_2$. Compute $u \xleftarrow{\$} \mathbb{Z}_q$ and send $\beta = ([u]P_1, [u](X + C_1))$ to the user.

$\mathsf{Unblind}_{\mathsf{BS}}(\beta, \mathsf{St}_{\mathsf{BS}}, \mathsf{pk}_{\mathsf{BS}})$**:** Parse β as (β_1, β_2), $\mathsf{St}_{\mathsf{BS}}$ as (m, t), $\mathsf{pk}_{\mathsf{BS}}$ as $(\mathsf{pk}_{\mathsf{CL}}, \cdot, \cdot)$ and compute $\sigma = (\beta_1, \beta_2 - [t]\beta_1)$. Return $\perp$ if $\mathsf{Verify}_{\mathsf{BS}}(m, \sigma, \mathsf{pk}_{\mathsf{BS}}) = 0$; otherwise return σ.

$\mathsf{Verify}_{\mathsf{BS}}(m, \sigma, \mathsf{pk}_{\mathsf{BS}})$**:** Parse $\mathsf{pk}_{\mathsf{BS}}$ as $(\mathsf{pk}_{\mathsf{CL}}, \cdot, \cdot)$ and return 1 iff $\mathsf{Verify}_{\mathsf{CL}}(m, \sigma, \mathsf{pk}_{\mathsf{CL}}) = 1$ and $e(Y_1, P_2) = e(P_1, Y_2)$ and $e(\hat{P}_1, Y_2) = e(\hat{Y}_1, P_2)$.

Scheme 1: Our Blind Signature Scheme

4 The MULTOS Card API and its Limitations

Most cryptographic algorithms rely on computations in finite groups, e.g. multiplicative group of integers modulo prime or additive groups on elliptic curves. In this section we investigate the operations that are supported by the MULTOS standard and their limitations.

Operation Name	Description	Supported on Our Card	Limitations	Function in API
Modular addition	$a + b \mod p$	No	-	Not supported by standard.
Addition	$a + b$	Yes	Works on blocks of bytes	`multosBlockAdd`
Modular subtraction	$a - b \mod p$	No	-	Not supported by standard.
Subtraction	$a - b$	Yes	Works on blocks of bytes	`multosBlockSubtract`
Modular multiplication	$a \cdot b \mod p$	Yes	-	`multosModularMultiplication`
Modular reduction	$b = a \mod p$	Yes	-	`multosModularReduction`
Modular inversion	$b = a^{-1} \mod p$	No	Not required by standard and not implemented on most available smart cards.	Requires low-level MEL primitive.
Modular exponentiation	$c = a^b \mod p$	Yes	Requires modulus of standard RSA bit length, i.e. 512, 1024 etc.	`multosModularExponentiation`
Modulo square roots	$b = \sqrt{a} \mod p$	No	-	-

Table 1. MULTOS API Operations Supporting Multiplicative Groups of Integers Modulo Prime p

Operation Name	Description	Supported by Our Card	Limitations	Function in API
Point Addition	$(x_R, y_R) = (x_P, y_P) + (x_Q, y_Q)$	No	Not required by the standard and not implemented on most available smart cards.	Requires low-level MEL primitive.
Point Multiplication	$(x_R, y_R) = [k](x_P, y_P)$	No	Not required by the standard and not implemented on most available smart cards.	Requires low-level MEL primitive.
EC Diffie-Hellman	$x_R = ([k](x_P, y_P))_x$	Yes	-	Requires low-level MEL primitive.

Table 2. MULTOS API Operations Supporting Elliptic Curves

4.1 Implementing the Missing Operations

In this subsection we present how to implement the operations that are missing on our card using the given operations. We begin by defining a structure that will hold all elements and the prime modulus of the multiplicative group, i.e.:

```
typedef struct {
    BYTE value[FIELD_LEN];
} FieldElement;
```

. Moreover, we will use the structure

```
typedef struct {
FieldElement x;
FieldElement y;
} ECPoint;
```

to represent elliptic curve points, the structure

```
typedef struct {
    BYTE format;
    BYTE length;
FieldElement p;
FieldElement a;
FieldElement b;
ECPoint G;
FieldElement q;
BYTE h;
} domainParameters;
```

to represent parameters of ordinary elliptic curves and the structures

```
typedef struct {
    ECPoint pub;
    FieldElement priv;
} ECKeyPair;

typedef struct {
    BYTE value[FIELD_LEN+4];
} ECDH_Secret;
```

to represent EC Diffie-Hellman keypairs and secrets.

Modular Addition and Subtraction We begin with basic operations, i.e. modular addition and modular subtraction. As one can see from Table 1, the MULTOS API supports standard addition and subtraction on blocks of bytes and we simple use those functions to implement the functions:

```
void mod_add(FieldElement a,FieldElement b,FieldElement modulo, FieldElement* result)
```

and

```
void mod_sub(FieldElement a,FieldElement b,FieldElement modulo, FieldElement* result)
```

Note that the implementation always ensures that `FieldElement` is always an element of the group and smaller than the prime modulus p. Thus, we can use the block addition method and check whether the result is smaller than p, otherwise we just subtract the modulus p from the result. Modular subtraction can be done in a similar way.

Modular Multiplication It is also easy to implement modular multiplication on `FieldElement` as:

```
void mod_mul(FieldElement a,FieldElement b,FieldElement modulo, FieldElement* result)
{
    multosModularMultiplication(FIELD_LEN,modulo.value,a.value,b.value);
    multosBlockCopyNonAtomic(FIELD_LEN,a.value,result->value);
}
```

To simplify operations on elliptic curves, we also implement function for computing modular squares

```
void mod_sq(FieldElement a,FieldElement modulo, FieldElement* result)
{
    multosModularMultiplication(FIELD_LEN,modulo.value,a.value,a.value);
    multosBlockCopyNonAtomic(FIELD_LEN,a.value,result->value);
}
```

and cubes

```
void mod_cube(FieldElement a,FieldElement modulo, FieldElement* result)
{
    multosBlockCopyNonAtomic(FIELD_LEN,a.value,c.value);
    multosModularMultiplication(FIELD_LEN,modulo.value,c.value,a.value);
    multosModularMultiplication(FIELD_LEN,modulo.value,c.value,a.value);
    multosBlockCopyNonAtomic(FIELD_LEN,c.value,result->value);
}
```

Modular Exponentiation As one can see from Table 1 the `multosModularExponentiation` requires that the used modulus has a standard RSA length. This causes some problems as for our purposes (working on elliptic curves) we use a much smaller modulus, e.g. 256-bit. To solve this problem, we extend the 256-bit modulus p by computing a modulus $n = p \cdot (2^{255}+1)$, which can be used. Finally, we compute $r = g^x \mod n$ and reduce the result $r \mod p$ using the `multosModularReduction` function. Note that this in fact is the correct result, as there exists a k such that $g^x = r + k \cdot n = r + k \cdot p \cdot (2^{255}+1)$. Thus, $g^x \mod p = r \mod p + 0$. The source code is given below:

```
void mod_exp(FieldElement a,FieldElement exp,FieldElement modulo, FieldElement* result)
{
    multosBlockCopyNonAtomic(FIELD_LEN,modulo.value,mod64);
    multosBlockCopyNonAtomic(FIELD_LEN,a.value,in64+(64-FIELD_LEN));

    multosBlockAdd(FIELD_LEN,mod64+(64-FIELD_LEN),modulo.value,mod64+(64-FIELD_LEN));

    multosModularExponentiation(FIELD_LEN,64,exp.value,mod64,in64,out64);

    multosModularReduction(64,FIELD_LEN,out64,modulo.value);

    multosBlockCopyNonAtomic(FIELD_LEN,out64+(64-FIELD_LEN),result->value);

    //clearing auxiliary arrays
    multosBlockClear(64,mod64);
    multosBlockClear(64,in64);
    multosBlockClear(64,out64);
}
```

Modular Inversion Usually modular inversion is implemented using the extended euclidean algorithm, as this is one of the most efficient ways to compute an inverse. However, since the MULTOS card is running a virtual machine, the software implementation of this algorithm is less efficient than using modular exponentiation and the Fermat's little theorem. The idea is that we first compute the exponent $e = p - 2$ and compute the inverse of a modulo p by computing $a^e \mod p$.

```
void mod_inv(FieldElement a,FieldElement modulo,FieldElement* result)
{
    multosBlockCopyNonAtomic(FIELD_LEN,modulo.value,exponent.value);
    multosBlockDecrement(FIELD_LEN,exponent.value);
    multosBlockDecrement(FIELD_LEN,exponent.value);
    multosBlockCopyNonAtomic(FIELD_LEN,modulo.value,mod64);
```

```
    multosBlockCopyNonAtomic(FIELD_LEN,a.value,in64+(64-FIELD_LEN));
    multosBlockAdd(FIELD_LEN,mod64+(64-FIELD_LEN),modulo.value,mod64+(64-FIELD_LEN));
    multosModularExponentiation(FIELD_LEN,64,exponent.value,mod64,in64,out64);
    multosModularReduction(64,FIELD_LEN,out64,modulo.value);
    multosBlockCopyNonAtomic(FIELD_LEN,out64+(64-FIELD_LEN),result->value);

    //clearing auxiliary arrays
    multosBlockClear(64,mod64);
    multosBlockClear(64,in64);
    multosBlockClear(64,out64);
}
```

Modulo Square Roots To efficiently compute square roots modulo prime p we restrict the implementation to $p \equiv 3 \mod 4$. In such a case, the computation of a square root of the value a is fairly efficient. The resulting root can be computed as $b = a^{(p+1)/4} \mod p$. It is easy to see that:

$$\begin{aligned} b^2 \mod p &= (a^{(p+1)/4})^2 \mod p = a^{(p+1)/2} \mod p \\ &= a^{((p-1)+2)/2} \mod p = (a^{1/2})^{p-1} \cdot a^{2/2} \mod p = 1 \cdot a \mod p = a \mod p. \end{aligned}$$

The source code for this method is given below.

```
void mod_sqrt(FieldElement a,FieldElement modulo,FieldElement* result)
{ //only for p = 3 \mod 4
    multosBlockCopyNonAtomic(FIELD_LEN,modulo.value,tmp);
    multosBlockIncrement(FIELD_LEN,tmp);
    multosBlockShiftRight(FIELD_LEN,2,tmp,exponent.value);
    mod_exp(a,exponent,modulo,result);
}
```

Elliptic Curve Point Addition Above we defined all operations in multiplication groups of integers modulo a prime. Thus, we can define elliptic curves over such groups and use the definition of point addition as described in Definition 1. Given two distinct points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, first we compute the slope $\lambda = (y_Q - y_P)/(x_Q - x_P) \mod p$ and then the resulting points coordinates $x_R = \lambda^2 - x_P - x_Q \mod p$ and $y_R = \lambda(x_P - x_R) - y_P \mod p$. This can be done as shown in the source code below.

```
void pointAddition(domainParameters domain, ECPoint P, ECPoint Q, ECPoint* R){
  if(cmp(P.x,Q.x)==0)
  {
    if(cmp(P.y,Q.y)==0)
    { //point double
      mod_sq(P.x,domain.p,&tmp1);
      mod_add(tmp1,tmp1,domain.p,&tmp2);
      mod_add(tmp2,tmp1,domain.p,&tmp3); //3xp^2
      mod_add(tmp3,domain.a,domain.p,&tmp1); // a+3xp^2
      mod_shl(P.y,domain.p,&tmp2);
      mod_inv(tmp2,domain.p,&tmp3); //1/2P.y
      mod_mul(tmp3,tmp1,domain.p,&lambda);

      mod_sq(lambda,domain.p,&tmp1);
      mod_shl(P.x,domain.p,&tmp2);
      mod_sub(tmp1,tmp2,domain.p,&tmp3); // R.x
      multosBlockCopyNonAtomic(FIELD_LEN,tmp3.value,R->x.value);
```

```
      mod_sub(P.x,R->x,domain.p,&tmp1);
      mod_mul(lambda,tmp1,domain.p,&tmp2);
      mod_sub(tmp2,P.y,domain.p,&tmp3);
      multosBlockCopyNonAtomic(FIELD_LEN,tmp3.value,R->y.value);
      }
    else
    {
      ;//infinity
    }
  }
  else{
    mod_sub(Q.y,P.y,domain.p,&tmp1);
    mod_sub(Q.x,P.x,domain.p,&tmp2);

    mod_inv(tmp2,domain.p,&tmp3);

    mod_mul(tmp1,tmp3,domain.p,&lambda); // \lambda = (Q.y-P.y)/(Q.x-P.x) \mod p
    mod_sq(lambda,domain.p,&tmp1);
    mod_sub(tmp1,P.x,domain.p,&tmp2);
    mod_sub(tmp2,Q.x,domain.p,&tmp3);
    multosBlockCopyNonAtomic(FIELD_LEN,tmp3.value,R->x.value); // R.x = \lambda^2-P.x-Q.x

    mod_sub(P.x,tmp3,domain.p,&tmp1);
    mod_mul(tmp1,lambda,domain.p,&tmp2);
    mod_sub(tmp2,P.y,domain.p,&tmp3);
    multosBlockCopyNonAtomic(FIELD_LEN,tmp3.value,R->y.value); // R.y = \lambda*(P.x-R.x)-P.y
  }
}
```

Elliptic Curve Point Multiplication As one can see from Table 2, point multiplication is not available on our MULTOS card and we can only compute the result of the EC Diffie-Hellman protocol, which is in fact the x-coordinate of the result of a point multiplication. We can use this function to compute the resulting x-coordinate but we must reconstruct the y-coordinate. This can be done directly by solving the curves equation $y^2 = x^3 + ax + b \mod p$. However, this way we receive two solutions, i.e. y and $-y$. To solve this problem we can again use the EC Diffie-Hellman primitive as follows.

Let us assume that we want to compute $P = [k]G$, where k is the known scalar and G is a known point. We first compute the x-coordinate of P using the EC Diffie-Hellman primitive. Then we use the curves equation to compute two candidate points $P_1 = (x, y)$ and $P_2 = (x, -y)$. Note that $P = P_i$, for $i = 1$ or $i = 2$. We then compute $x_Q = ([k + 1]G)_x$ using the EC Diffie-Hellman primitive and $(x_{Q_1}, y_{Q_1}) = Q_1 = P_1 + G$ using the above point addition function. We the check whether $x_Q = x_{Q_1}$, if so then $P = P_1$ and $P = P_2$ otherwise. The source code for the EC Diffie-Hellman primitive and the above technique is given below.

```
void ECDH(domainParameters domain,FieldElement s,ECPoint pub,ECDH_Secret* R)
{
  __push (&domain);
  __push (s.value);
  __push (&pub);
  __push (R);
  __code (PRIM, DH, 0x00);
}

void pointMultiplication(domainParameters domain,FieldElement s,ECPoint pub,ECPoint* R)
```

```
{
  __push (&domain);
  __push (s.value);
  __push (&pub);
  __push (R);
  __code (PRIM, DH, 0x00);
  multosBlockCopyNonAtomic(FIELD_LEN,R->x.value,xd.value);

  multosBlockIncrement(FIELD_LEN,s.value);
  __push (&domain);
  __push (s.value);
  __push (&pub);
  __push (R);
  __code (PRIM, DH, 0x00);
  multosBlockCopyNonAtomic(FIELD_LEN,R->x.value,xd1.value);
  multosBlockCopyNonAtomic(FIELD_LEN,xd.value,R->x.value);
  multosBlockDecrement(FIELD_LEN,s.value);

  mod_mul(pub.x,xd,domain.p,&tmp1);
  mod_add(tmp1,domain.a,domain.p,&tmp2);
  mod_add(pub.x,xd,domain.p,&tmp3);
  mod_mul(tmp2,tmp3,domain.p,&tmp1); //  (x+xd)(a+x*xd)

  mod_shl(domain.b,domain.p,&tmp2); // 2b
  mod_add(tmp2,tmp1,domain.p,&tmp3); //  2b + (x+xd)(a+x*xd)

  mod_sub(pub.x,xd,domain.p,&tmp1);
  mod_sq(tmp1,domain.p,&tmp2); //  (x-xd)^2
  mod_mul(tmp2,xd1,domain.p,&tmp1); //  xd1*(x-xd)^2
  mod_sub(tmp3,tmp1,domain.p,&tmp2); // 2b + (x+xd)(a+x*xd) - xd1*(x-xd)^2

  mod_shl(pub.y,domain.p,&tmp1); //   2y
  mod_inv(tmp1,domain.p,&tmp3); // 1/2y
  mod_mul(tmp3,tmp2,domain.p,&tmp1); // result!

  multosBlockCopyNonAtomic(FIELD_LEN,tmp1.value,R->y.value);
}
```

Generating Random Elliptic Curve Key Pairs In order to generate random EC key pairs we use the following function.

```
void generateKeyPair(domainParameters domain,ECPoint* P,FieldElement* scalar)
{
  ECKeyPair pair;
  __push (&domain);
  __push (&pair);
  __code (PRIM, PRIM_GEN_ECC_PAIR, 0x00);
  multosBlockCopyNonAtomic(FIELD_LEN,pair.pub.x.value,P->x.value);
  multosBlockCopyNonAtomic(FIELD_LEN,pair.pub.y.value,P->y.value);
  multosBlockCopyNonAtomic(FIELD_LEN,pair.priv.value,scalar->value);
}
```

5 Implementing User Side of the Blind Signature Scheme

We begin this section with the observation that beside the signature verification $\mathsf{Verify}_{\mathsf{BS}}(m, \sigma, \mathsf{pk}_{\mathsf{BS}})$ (which requires the pairing operation and operations in $\mathbb{G}_2$) in the $\mathsf{Unblind}_{\mathsf{BS}}(\beta, \mathsf{St}_{\mathsf{BS}}, \mathsf{pk}_{\mathsf{BS}})$ algorithm, the user only performs operations in the group $\mathbb{G}_1$. As noted earlier, the required groups can be instantiated by BN-curves [12]). In such a case, the group $\mathbb{G}_1$ is an ordinary elliptic curve. On the other hand, the group $\mathbb{G}_2$ is an elliptic curve defined over an extension field and our tests have shown that the efficiency of such operations is unsatisfactory.

Fortunately, a signature verification does not require any secret values and it can be easily delegated by the card to the reader. It follows that we can use the operations described in Section 4 to implement the user side of the blind signature scheme.

$\mathsf{Request}_{\mathsf{BS}}(m, \mathsf{pk}_{\mathsf{BS}})$ Algorithm

We will now show how to compute the output $\rho = (\rho_1, \rho_2)$. At first, we must create variables to store the elliptic curve points $P_1, Y_1, \hat{P}_1, \hat{Y}_1$.

```
ECPoint P1;
ECPoint HP1;
ECPoint Y1;
ECPoint HY1;
```

and send those values to the smart card. Those values can be initialized using the `multosBlockCopyNonAtomic` function. Moreover, we must create variables to store the output ρ, which consists of two elliptic curve points, and the state $\mathsf{St}_{\mathsf{BS}}$, which consists of two group elements.

```
ECPoint RHO1;
ECPoint RHO2;
FieldElement m;
FieldElement t;
```

We also use auxiliary values

```
ECKeyPair TMP;
ECPoint Q1;
ECPoint Q2;
ECPoint Q3;
```

and variable

```
domainParameters domain;
```

to store the domain parameters for group $\mathbb{G}_1$.

Finally, we compute the output using the following commands

```
generateKeyPair(domain,&TMP.pub,&TMP.priv); // TMP.pub contains [t]P_1
pointMultiplication(domain,m,Y1,&Q1); // Q1 contains [m]Y_1
pointAddition(domain,TMP.pub,Q1,&RHO1); // RHO1 contains [t]P_1+[m]Y_1

pointMultiplication(domain,TMP.priv,HP1,&Q2); // Q2 contains [t]\hat{P}_1
pointMultiplication(domain,m,HY1,&Q3); // Q3 contains [m]\hat{Y}_1
pointAddition(domain,Q1,Q2,&RHO2); // RHO2 contains [t]\hat{P}_1+[m]\hat{Y}_1
```

Note that the state value $\mathsf{St}_{\mathsf{BS}}$ is (m, t).

$\mathsf{Unblind}_{\mathsf{BS}}(\beta, \mathsf{St}_{\mathsf{BS}}, \mathsf{pk}_{\mathsf{BS}})$ Algorithm

We will now show how to compute the final signature $\sigma^* = (\sigma_1^*, \sigma_2^*)$. This time we must create two variables to store the input $\beta = (\beta_1, \beta_2)$, the final signature σ^* and a the value $-t \mod q$

```
ECPoint S1;
ECPoint S2;
ECPoint Beta1;
ECPoint Beta2;
FieldElement tt;
```

To compute the signature σ^* we use the following source code. First we compute the value $-t \mod q$ as

```
mod_sub(domain.q,t,domain.q,&tt); // tt contains -t \mod q
```

Then we compute

```
multosBlockCopyNonAtomic(FIELD_LEN,Beta1->x.value,S1->x.value);
multosBlockCopyNonAtomic(FIELD_LEN,Beta1->y.value,S1->y.value); // S1 contains Beta1
pointMultiplication(domain,tt,Beta1,&TMP); // TMP contains [-t]\beta_1
pointAddition(domain,Beta2,TMP,&S2); // S2 contains \beta_2+[-t]\beta_1
```

and send the values `m`, `S1` and `S2` to the reader for verification. If the signature is valid, then it can be processed by other algorithms, e.g. anonymous credential or e-coin applications, which use blind signatures as a component.

Conclusions

In this report we presented a way to efficiently implement the user side of a practical two-move blind signature. The described results can be used to implement various other cryptographic algorithms and protocols on MULTOS smart cards.

Acknowledgements

This research was supported by the National Science Centre (Poland) based on decision no 2014/15/N/ST6/04577.

References

1. Chaum, D.: Blind Signatures for Untraceable Payments. In: Advances in Cryptology: Proceedings of CRYPTO '82, Plenum (1982) 199–203
2. Fischlin, M.: Round-Optimal Composable Blind Signatures in the Common Reference String Model. In Dwork, C., ed.: Advances in Cryptology - CRYPTO 2006. Volume 4117 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 60–77
3. Fischlin, M., Schrder, D.: On the Impossibility of Three-Move Blind Signature Schemes. In Gilbert, H., ed.: Advances in Cryptology EUROCRYPT 2010. Volume 6110 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 197–215
4. Garg, S., Rao, V., Sahai, A., Schrder, D., Unruh, D.: Round Optimal Blind Signatures. In Rogaway, P., ed.: Advances in Cryptology CRYPTO 2011. Volume 6841 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 630–648
5. Garg, S., Gupta, D.: Efficient Round Optimal Blind Signatures. In Nguyen, P., Oswald, E., eds.: Advances in Cryptology EUROCRYPT 2014. Volume 8441 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2014) 477–495
6. Fuchsbauer, G., Hanser, C., Slamanig, D.: Practical Round-Optimal Blind Signatures in the Standard Model. Cryptology ePrint Archive, Report 2015/626 (2015) `http://eprint.iacr.org/`.
7. Fuchsbauer, G., Hanser, C., Kamath, C., Slamanig, D.: Practical Round-Optimal Blind Signatures in the Standard Model from Weaker Assumptions. Cryptology ePrint Archive, Report 2016/662 (2016) `http://eprint.iacr.org/2016/662`.

8. Fuchsbauer, G., Hanser, C., Slamanig, D.: EUF-CMA-Secure Structure-Preserving Signatures on Equivalence Classes. Cryptology ePrint Archive, Report 2014/944 (2014) http://eprint.iacr.org/.
9. Pointcheval, D., Sanders, O.: Short randomizable signatures. Cryptology ePrint Archive, Report 2015/525 (2015) http://eprint.iacr.org/.
10. Bellare, M., Palacio, A.: The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols. In: Advances in Cryptology - CRYPTO 2004, Santa Barbara, California, USA, August 15-19, 2004, Proceedings. Volume 3152 of Lecture Notes in Computer Science., Springer (2004) 273–289
11. Hanzlik, L., Kluczniak, K.: A Short Paper on Blind Signatures from Knowledge Assumptions. Financial Cryptography 2016 - preproceedings (2016) http://fc16.ifca.ai/preproceedings/31_Hanzlik.pdf.
12. Barreto, P.S.L.M., Naehrig, M.: Pairing-Friendly Elliptic Curves of Prime Order. In Preneel, B., Tavares, S.E., eds.: Selected Areas in Cryptography. Volume 3897 of Lecture Notes in Computer Science., Springer (2005) 319–331