A Fiat-Shamir Implementation Note

Simon Cogliani, Rémi Géraud, and David Naccache

École normale supérieure,
PSL Research University, Paris, France.
given_name.family_name@ens.fr

Abstract. In the Micali-Shamir paper [7] improving the efficiency of the original Fiat-Shamir protocol [5, 6, 9], the authors state that

> *"(...) not all of the v_i's will be quadratic residues mod n. We overcome this technical difficulty with an appropriate perturbation technique (...)"*

This perturbation technique is made more explicit in the associated patent application [8]:

> *"Each entity is allowed to modify the standard v_j which are QNRs. A particularly simple way to achieve this is to pick a modulus $n = pq$ where $p = 3 \bmod 8$ and $q = 7 \bmod 8$, since then exactly one of $v_j, -v_j, 2v_j, -2v_j$ is a QR mod n for any v_j. The appropriate variant of each v_j can be (...) deduced by the verifier himself during the verification of given signatures."*

In this short note we clarify the way in which the verifier can infer by himself the appropriate variant of each v_j during verification.

1 Introduction

The increased popularity of lightweight implementations invigorates the interest in the resource-preserving protocols of the late 1980s initially designed for smart-cards. By then, cryptoprocessors were expensive and cumbersome, hence the research community started looking for astute ways to identity and sign with scarce resources.

One such particularly elegant procedure is the the Fiat-Shamir protocol [5] (that we do not fully restate here). In the original procedure, the public-keys are derived as follows [6, 9]:

> *(...) the center (...) chooses and makes public (...) a PRF f which maps arbitrary strings to the range $[0, n)$. (...) The center then performs the following steps:*
>
> 1. *Compute the values $v_j = f(I, j)$ for small values of j.*
> 2. *Pick distinct k values of j for which v_j is a QR mod n and compute the smallest square root s_j of v_j^{-1}.*
> 3. *Issue a smart card which contains I, the k s_j values, and their indices.*

In a follow-up paper by Micali and Shamir [7], the authors propose a way to reduce the verifier's work factor by selecting small public-keys. In this version, the v_j's are the first small primes[1]. However, as noted in [7], doing so does not yield only QRs:

> *"(...) not all of the v_i's will be quadratic residues mod n. We overcome this technical difficulty with an appropriate perturbation technique (...)"*

The associated patent [8] describes further the idea:

> *"Each entity is allowed to modify the standard v_j which are QNRs. A particularly simple way to achieve this is to pick a modulus $n = pq$ where $p = 3 \bmod 8$ and $q = 7 \bmod 8$, since then exactly one of $v_j, -v_j, 2v_j, -2v_j$ is a QR mod n for any v_j. The appropriate variant of each v_j can be (...) deduced by the verifier himself during the verification of given signatures."*

Indeed, we have the following well-known result:

Lemma 1. *For $n = pq$ where $p = 3 \bmod 8$ and $q = 7 \bmod 8$ (such moduli are sometimes known as Williams numbers), -1 and 2 are both quadratic non-residues modulo n.*

Proof. For -1 to be a quadratic residue mod n, it has to be a quadratic residue modulo every prime that divides n, i.e. it has to be a QR modulo p and q. One easily checks that

$$\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2} = -1 \quad \text{and} \quad \left(\frac{-1}{q}\right) = (-1)^{(q-1)/2} = -1$$

because both p and q are equal to -1 modulo 4. Similarly,

$$\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8} = -1$$

because $n = pq = -3 \bmod 8$. Thus both -1 and 2 are QNR mod n. □

However there is no indication, in the paper nor in the associated patent, as to how exactly the verifier can deduce which or the four possibilities should be considered.

1.1 Existing approaches

Besides leaving the verifier to determine which v_j are QR, [8] mentions providing this information explicitly to the verifier, either alongside the public key material, or during the protocol. It also points out that using $d = 3$ and n such that

[1] The choice of the v_j as the first k primes is motivated by the fact that large values make the scheme less efficient, and the observation that multiplicatively related values can make the scheme less secure. More generally, the v_j can be relatively prime small integers with small Hamming weight.

$\phi(n) \nmid 3$ so that every v_j has a cubic root mod n. However the choice of such a d exposes the participants to Wiener's attack [2–4,11], and in any case requires an additional modular multiplication during the generation and the verification of signatures.

When the Micali-Shamir scheme is considered, the question of the QNR is usually evacuated by making sure that only v_j that are QR are part of the public key (see e.g. [1]). The downside of such an approach is that many values of v_j cannot be chosen, and since the v_j are prime this causes an increase in the public key size, and an overall loss of efficiency.

2 Compensating coefficients

The way in which the verifier can deduce which ϵv_j (where $\epsilon \in \{-2, -1, 1, 2\}$) to use is left unexplicited in [8], but explicited in [10]. But it can be done as follows. We use the equivalent values $v_j, -v_j, v_j/2, -v_j/2$ for the sake of faster calculations.

Denote by $\epsilon_i \in \{-1/2, -1, 1, 1/2\}$ the value such that $\epsilon_i v_i$ is a QR mod n. The prover keeps two binary strings α, β defined as follows:

$$\alpha_i = \begin{cases} 0 & \text{if } |\epsilon_i| = 1 \\ 1 & \text{if } |\epsilon_i| = 1/2 \end{cases}$$

$$\beta_i = \begin{cases} 0 & \text{if the sign of } \epsilon_i \text{ is } + \\ 1 & \text{if the sign of } \epsilon_i \text{ is } - \end{cases}$$

For a given challenge e we define:

$$u = \sum_{i=0}^{k-1} \alpha_i e_i \quad \text{and} \quad w = \sum_{i=0}^{k-1} \beta_i e_i \bmod 2$$

and $\overline{u} = u \operatorname{div} 2$ and $\underline{u} = u \bmod 2$ (so that in particular $u = 2\overline{u} + \underline{u}$).

We now describe three approaches that allow the verifier to perform its task.

2.1 Version 1: The prover sends u and w

Because not all the v_j are QR, when computing directly with v_j the verification algorithm must check that:

$$y^2 \prod_{i=0}^{k-1} v_j^{e_j} = (-1)^w 2^u x \bmod n.$$

This is easy to verify, provided that the prover has sent u and w alongside their response. This requires the transmission of a few bits (u and w can typically be encoded using a single byte). Note that u and w are computed from the (public) values of the v_j and of e, so that there is no information leaked in sharing these numbers.

2.2 Version 2: No correction

In fact, it is not necessary to transmit u or w. Indeed, from

$$y^2 \prod_{i=0}^{k-1} v_j^{e_j} = (-1)^w 2^u x \bmod n.$$

The verifier can compute:

$$\Delta = x^{-1} y^2 \prod_{i=0}^{k-1} v_j^{e_j} \bmod n$$

because $2^u < 2^k < n$. The verifier therefore only needs to check that either Δ or $n - \Delta$ is of the form 2^u in $\mathbb{Z}$, for some u. This of course can be checked very efficiently. While minimizing the prover's effort, note that this variant requires from the verifier an extra modular inversion.

Note also that the tolerance concerning extra -1 and 2 in the verification formula cannot be confused with the use of v_j equal to these values as both -1 and 2 are QNRs. Therefore we do not impact the protocol's soundness.

2.3 Version 3: The prover compensates $\overline{u}$

In this approach, we alter the definition of y, which is now:

$$y = 2^{\overline{u}} r \prod_{i=0}^{k-1} s_j^{e_j}.$$

We call this operation "compensating $\overline{u}$". The computation of y is performed by the prover. Then the verifier can check that:

$$\begin{aligned} \Gamma &= 2y^2 \prod_{i=0}^{k-1} v_j^{e_j} \\ &= 2 \left(2^{\overline{u}} r \prod_{i=0}^{k-1} s_j^{e_j} \right)^2 \times \prod_{i=0}^{k-1} v_j^{e_j} \\ &= (-1)^w 2^{\neg \underline{u}} x \bmod n. \end{aligned}$$

In other words, all the verifier has to do is check if

$$\Gamma \in \{x, n - x, 2x \bmod n, -2x \bmod n\}.$$

This verification is quick and easy: Start by comparing to x or $n - x$. One of the values $x, n - x$ is a number ℓ by one bit shorter than n. A simple shift to the right of ℓ therefore allows to continue comparing (subtract n again if needed).

Alternatively, the two bits $\neg \underline{u}, w$ can be sent to the verifier to further speed-up verification. Note here again that these quantity do not leak any secret information.

3 Security analysis

In the constructions of Sections 2.1 and 2.2 only the verifier's algorithm is modified, and it is straightforward to see that the verifier will not accept with our modifications a response that they would not have accepted using the original Fiat-Shamir verification algorithm.

However the variant of Section 2.3 proposes a different definition of y, and we must show that this does not impact the scheme's security. Note that soundness is guaranteed from the observation that -1 and 2 are QNR mod n, so that no new valid response is introduced by altering the verification procedure in the way we did. There remains to show that the honest-verifier zero-knowledge property still holds, by expliciting a simulator. This is straightforward when $\overline{u}$ is public (just multiply by $2^{\overline{u}}$ the output y of a Fiat-Shamir simulator). When $\overline{u}$ is not public, the simulator can do the same after drawing a value $\overline{u}$ at random.

Nevertheless, these arguments only show that our modifications do not impact the security of the Micali-Shamir variant of the Fiat-Shamir protocol, not that it is secure in the first place. The discussion in [7] gives a heuristic argument, and refers to a full version of the paper that, to the best of our knowledge, never appeared. Bellare and Ristov [1] call this claim the "square roots of prime products" assumption.

4 Toy Examples

The following notebook (written in Sage version 7.2) demonstrates the three techniques on a toy example.

```
# Initialization processus
# Number of elements to check
k = 10

# Williams numbers with primes with 24 bits each.
p = 32452759
q = 32452843
n = p*q

def setup_public_key(v, s, alpha, beta, inverse_modular=True):
    coeff = [-2, -1, 1, 2]

    for i in range(k):
        val = random_prime(2^24-1, False, 2^23)
        v += [val%n]

        # Jacobi symbol for checking if val is a QR
        # if not, we try the perturbation technique
        for j in range(len(coeff)):
            sigma = coeff[j]
            if kronecker_symbol(sigma*val, p) == 1 and kronecker_symbol(sigma*val, q) == 1:
                break

        if abs(sigma) == 1:
            alpha += [0]
        else:
            alpha += [1]

        if sigma > 0:
            beta += [0]
```

```
            else:
                beta += [1]

            if inverse_modular:
                if abs(sigma) == 2:
                    sigma = inverse_mod(sigma, n)

            val *= sigma

            # Lagrange tricks and Chinese Remainder Theorem for finding the square root modulo n
            s2 = inverse_mod(val, n)
            s += [crt(s2^((p+1)/4), s2^((q+1)/4), p, q)]

    # The prover sends u and w
    def version1():
        v = []
        s = []
        alpha = []
        beta = []

        setup_public_key(v, s, alpha, beta)

        ## Fiat-Shamir Sigma Protocol
        # Commitment
        r = randint(1,n)
        x = mod(r^2, n)

        # Challenge
        e = [randint(0,1) for i in range(k)]

        # Response
        y = r*prod((s[j])^e[j] for j in range(k))
        u = sum(alpha[i]*e[i] for i in range(k))
        w = mod(sum(beta[i]*e[i] for i in range(k)), 2)

        # Verifier Checking
        print y^2*prod(v[i]^e[i] for i in range(k)) == x*(-1)^(w)*(2)^(u)

    # No correction
    def version2():
        v = []
        s = []
        alpha = []
        beta = []

        setup_public_key(v, s, alpha, beta)

        ## Fiat-Shamir Sigma Protocol
        # Commitment
        r = randint(1,n)
        x = r^(2)%n

        # Challenge
        e = [randint(0,1) for i in range(k)]

        # Response
        y = r*prod((s[j])^e[j] for j in range(k))
        u = sum(alpha[i]*e[i] for i in range(k))
        w = mod(sum(beta[i]*e[i] for i in range(k)), 2)

        # Verifier Checking
        delta = mod(inverse_mod(x, n)*y^2*prod(v[i]^e[i] for i in range(k)), n)

        try:
            print log(delta, 2) in ZZ
        except ValueError:
            print log(n-delta, 2) in ZZ
```

```
# The prover compensates $\overline{u}$
def version3():
    v = []
    s = []
    alpha = []
    beta = []

    setup_public_key(v, s, alpha, beta)

    ## Fiat-Shamir Sigma Protocol
    # Commitment
    r = randint(1,n)
    x = mod(r^(2), n)

    # Challenge
    e = [randint(0,1) for i in range(k)]

    # Response
    u = sum(alpha[i]*e[i] for i in range(k))
    w = mod(sum(beta[i]*e[i] for i in range(k)), 2)
    u_overline = u >> 1
    u_underline = u % 2

    y = 2^(u_overline)*r*prod((s[j])^e[j] for j in range(k))

    Gamma = mod(-1^w * 2^(u_underline.__xor__(1)) * x, n)

    print Gamma == x or Gamma == n-x or Gamma == mod(2*x, n) or Gamma == mod(-2*x, n)
```

References

1. Bellare, M., Ristov, T.: Hash functions from sigma protocols and improvements to VSH. In: Pieprzyk, J. (ed.) Advances in Cryptology – ASIACRYPT 2008. Lecture Notes in Computer Science, vol. 5350, pp. 125–142. Springer, Heidelberg, Germany, Melbourne, Australia (Dec 7–11, 2008)
2. Blömer, J., May, A.: A generalized wiener attack on RSA. In: Bao, F., Deng, R., Zhou, J. (eds.) PKC 2004: 7th International Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 2947, pp. 1–13. Springer, Heidelberg, Germany, Singapore (Mar 1–4, 2004)
3. Boneh, D., Durfee, G.: Cryptanalysis of RSA with private key d less than $n^{0.292}$. In: Stern, J. (ed.) Advances in Cryptology – EUROCRYPT'99. Lecture Notes in Computer Science, vol. 1592, pp. 1–11. Springer, Heidelberg, Germany, Prague, Czech Republic (May 2–6, 1999)
4. Coppersmith, D.: Small solutions to polynomial equations, and low exponent RSA vulnerabilities. Journal of Cryptology 10(4), 233–260 (1997)
5. Feige, U., Fiat, A., Shamir, A.: Zero-knowledge proofs of identity. Journal of Cryptology 1(2), 77–94 (1988)
6. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) Advances in Cryptology – CRYPTO'86. Lecture Notes in Computer Science, vol. 263, pp. 186–194. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 1987)
7. Micali, S., Shamir, A.: An improvement of the Fiat-Shamir identification and signature scheme. In: Goldwasser, S. (ed.) Advances in Cryptology – CRYPTO'88. Lecture Notes in Computer Science, vol. 403, pp. 244–247. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 21–25, 1990)

8. Shamir, A.: Variants of the Fiat-Shamir identification and signature scheme (Jun 12 1990), https://www.google.ch/patents/US4933970, US Patent 4,933,970
9. Shamir, A., Fiat, A.: Method, apparatus and article for identification and signature (May 31 1988), https://www.google.com/patents/US4748668, US Patent 4,748,668
10. Simmons, G.J., Purdy, G.B.: Zero-knowledge proofs of identity and veracity of transaction receipts. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 35–49. Springer (1988)
11. Wiener, M.J.: Cryptanalysis of short RSA secret exponents (abstract). In: Quisquater, J.J., Vandewalle, J. (eds.) Advances in Cryptology – EUROCRYPT'89. Lecture Notes in Computer Science, vol. 434, p. 372. Springer, Heidelberg, Germany, Houthalen, Belgium (Apr 10–13, 1990)