An Implementation of Efficient Pseudo-Random Functions

Michael Langberg

March 25, 1998

Abstract

Naor and Reingold [3] have recently introduced two new constructions of very efficient pseudo-random functions, proven to be as secure as the decisional version of the Diffie-Hellman [2] assumption or the assumption that factoring is hard. In this site we define and exhibit an implementation of the construction which is based on the decisional Diffie-Hellman assumption.

1 Naor and Reingold's main construction

Naor and Reingold define a pseudo-random function $\tilde{f}_{P,Q,g,\vec{a}}$ as follows: The key of each pseudo-random function is a tuple, $\langle P, Q, g, \vec{a}\rangle$, where P is a large prime, Q a large prime divisor of $P-1$, g an element of order Q in $\mathbf{Z}_P^*$ and $\vec{a} = \langle a_0, a_1, \ldots a_n\rangle$ a uniformly distributed sequence of $n+1$ elements in $\mathbf{Z}_Q$. For any n-bit input, $x = x_1 x_2 \cdots x_n$, the function $\tilde{f}_{P,Q,g,\vec{a}}$ is defined by:

$$\tilde{f}_{P,Q,g,\vec{a}}(x) \stackrel{\text{def}}{=} (g^{a_0})^{\prod_{x_i=1} a_i} \bmod P$$

This function is the core of our implementation.

2 Implementation

2.1 Outline

The function $\tilde{f}_{P,Q,g,\vec{a}}$ has domain $\{0,1\}^n$ and range $\langle g\rangle$ (the subgroup of $\mathbf{Z}_P^*$ generated by g). We now show how to adjust $\tilde{f}_{P,Q,g,\vec{a}}$ in order to get a $\{0,1\}^* \to \{0,1\}^\ell$ pseudo-random function. I.e., the input of the function is a bit string of *arbitrary length* and the output is a pseudo-random bit string of fixed length ℓ (instead of a pseudo-random element in $\langle g\rangle$). For every input $x \in \{0,1\}^*$ the function $f = f_{P,Q,g,\vec{a},\hat{h},h} : \{0,1\}^* \to \{0,1\}^\ell$ we implement is defined by:

$$f(x) = h(\tilde{f}(\hat{h}(x), 0), \tilde{f}(\hat{h}(x), 1))$$

where $\hat{h}, h$ are two hash functions (defined below) and $\tilde{f} = \tilde{f}_{P,Q,g,\vec{a}}$. In words, the computation of f is as follows: First apply $\hat{h}$ on the input to get $y = \hat{h}(x)$. Then compute two pseudo-random elements $\tilde{f}(y,0)$ and $\tilde{f}(y,1)$ in $\langle g\rangle$. Finally concatenate these elements and hash the outcome by the second hash function h.

2.2 The Role and Definition of $\hat{h}$

The first step in the computation of f is hashing the input $x \in \{0,1\}^*$ to receive an element $y = \hat{h}(x)$ s.t. $(y,0)$ and $(y,1)$ are in the domain of the pseudo-random function $\tilde{f}$. In order for $\tilde{f}(\hat{h}(x),\cdot)$ to be pseudo-random it is enough to require that for any two different inputs $x, y \in \{0,1\}^*$ the probability of collision, $\Pr_{\hat{h}}[\hat{h}(x) = \hat{h}(y)]$, is negligible. To get this we define $\hat{h} = \hat{h}_r : \{0,1\}^* \to \mathbf{Z}_R$ as follows:

$$\hat{h}_r(m) = (\sum_{i=1}^{k} m_i \cdot r^{i-1}) \bmod R$$

where R is a 161-bit prime, r (the key of $\hat{h}$) is a uniformly distributed element in $\mathbf{Z}_R^*$ and the input m is partitioned into a sequence, $\langle m_1 \ldots m_k\rangle$, of elements in $\mathbf{Z}_R$. With this definition the collision probability on any two inputs of length at most $160k$ is bounded by $\frac{k}{R} \leq \frac{k}{2^{160}}$. The probability of collision for some pair of inputs among ℓ arbitrarily chosen inputs is bounded by $\frac{k\ell^2}{2^{160}}$. For practical values of ℓ and k this probability is sufficiently small.

2.3 The Role and Definition of h

As mentioned above, on input $y \in \{0,1\}^n$ our core function $\tilde{f}$ will output a pseudo-random element in the subgroup $\langle g\rangle$. Converting this element into a pseudo-random value in $\{0,1\}^\ell$ is done using a second hash function $h : (\mathbf{Z}_R)^k \to \mathbf{Z}_R$. The requirement from h is that for any pair of different inputs x, y the collision probability $\Pr_h[h(x) = h(y)]$ is $1/R$ (or "extremely close" to this value). Therefore, we cannot define $h = h_y$ in the same way we defined $\hat{h}$. Rather than that we use the following definition:

$$h_y(m) = (\sum_{i=1}^{k} m_i \cdot y_i) \bmod R$$

where R is a 161-bit prime (as above), $y = \langle y_1 \ldots y_k\rangle$ (the key of $\hat{h}$) is a uniformly distributed sequence of elements in $\mathbf{Z}_R$ and the input $m = \langle m_1 \ldots m_k\rangle$ is a sequence of elements in $\mathbf{Z}_R$.

With this definition of h we can conclude from [3] that if X is a random variable uniformly distributed over a set of size 2^{400}, then for all but a fraction 2^{-80} of the choices of y the random variable $h(X)$ is of statistical distance at most 2^{-80} from the uniform distribution over $\mathbf{Z}_R$. Therefore, if X is a pseudo-random element in such a set then (for all but a fraction 2^{-80} of the choices of y) the random variable $h(X)$ is a pseudo-random value in $\mathbf{Z}_R$. Note that choosing R extremely close to 2^{160} guarantees $h(X)$ to be pseudo-random in $\{0,1\}^{160}$. In our case, for any $x \in \{0,1\}^*$ the value $\tilde{f}(\hat{h}(x))$ is pseudo-random in $\langle g\rangle$. Thus if we fix $Q \approx 2^{400}$ (implying the size of $\langle g\rangle$ to be $\approx 2^{400}$) we can define our pseudo-random function as

$$f(x) = h(\tilde{f}(\hat{h}(x))) : \{0,1\}^* \to \{0,1\}^{160}$$

However, defining Q of size 400 bits seems to be an overkill (in terms of security) and leads to an inefficient implementation. We therefore use the following optimization: We define Q of size 200 bits and on input $x \in \{0,1\}^*$ we compute the element $\langle \tilde{f}(\hat{h}(x),0), \tilde{f}(\hat{h}(x),1)\rangle$ which is pseudo-random in $\langle g\rangle^2$. Since $\langle g\rangle^2$ is of size approximately 2^{400}, we define $f : \{0,1\}^* \to \{0,1\}^{160}$ (by the previous analysis) to be

$$f(x) = h(\tilde{f}(\hat{h}(x),0), \tilde{f}(\hat{h}(x),1))$$

This suggestion is more efficient than the previous one since the exponent of g in the computation of $\tilde{f}(\hat{h}(x),1)$ can be derived from the exponent of g in the computation of $\tilde{f}(\hat{h}(x),0)$ using a single modular multiplication.

2.4 The size of P and Q

In our discussion so far we assumed that $\tilde{f}$ is a pseudo-random function. As shown by Naor and Reingold, this is true as long as the decisional version of the Diffie-Hellman assumption holds. In the current state of knowledge it seems that choosing P of 1000 bits and Q of 200 bits makes this assumption sufficiently safe.

2.5 Constants

The specific constants used in our implementation are:

1. $Q = 2^{200} + 235$, $P = (Q^4 - 295) \cdot Q + 1$.
2. $g = 11^{\frac{P-1}{Q}}$.
3. $n = 160$.
4. $R = 2^{160} + 7$.

2.6 Large numbers :

For all computations and primality checks involving large numbers we used the NTL package [4] by V. Shoup.

3 Key Generation

In this implementation the key $P, Q, g, \vec{a}, \hat{h}, h$ was generated as follows:

The Triplet $\langle P, Q, g\rangle$

In order to define $\tilde{f}$, we need two large primes P, Q s.t. $Q|(P-1)$ and an element $g \in \mathbf{Z}_Q^*$ of order Q. This task is achieved in three steps. First we find a prime Q of size $\approx 2^{200}$, then we find a prime P of size $\approx 2^{1000}$ such that $P = \alpha Q + 1$ for some $\alpha \in \mathbf{N}$. The density of primes in the natural numbers allows us to find appropriate Q and α quite easily. Finally we fix g to be $\beta^{\frac{P-1}{Q}}$ for some $\beta \in \mathbf{N}$ such that $g \neq 1$ (the primality of Q ensures that the order of g is exactly Q). Note that we do not insist that P, Q and g be uniform in their range (since it is not clear that such a requirement is essential for the Diffie-Hellman assumption).

The Keys $\vec{a}$, r and y

In order to implement f a large amount of random (or pseudo-random) keys are needed. These keys can be generated by a pseudo-random generator which is a slight modification of our construction f. Let $\hat{f}_{P,Q,g,\hat{a},h} = f_{P,Q,g,a,Id,h}$ where $a = \langle a_0, a_1 \ldots a_n \rangle$, $\hat{a} = \langle 1, a_1 \ldots a_n \rangle$ and Id is the identity function. Following the proof in [3] it can be proven that $\hat{f}$ is pseudo-random on all inputs but 0. Using $\hat{f}$ we implement the pseudo-random function :

$$\hat{f}^{(1)}_{P,Q,g,\hat{a}_1,h} : \{0,1\}^3 \to \{0,1\}^{160}$$

This implementation itself uses a seed of 2880 random bits : 800 for the random key $\hat{a}_1$ and 2080 for the key y. Denoting $y_i = \hat{f}^{(1)}(i)$ for $i = 1 \ldots 7$ we have, by our above analysis, that each y_i is pseudo-random in $\{0,1\}^{160}$. Thus $\vec{y} = \langle y_1 \ldots y_7 \rangle$ is pseudo-random in $\{0,1\}^{1120}$. Using $\vec{y}$ as a random source we can derive a new pseudo-random key, $\hat{a_2}$, with 5 elements by partitioning $\vec{y}$ into chunks of length $\lfloor \log Q \rfloor$. Choosing Q such that $\frac{|Q-2^{\lfloor \log Q \rfloor}|}{Q}$ is negligible guarantees that the elements received remain pseudo-random in $\mathbf{Z}_Q$. Our new key now allows us to define

$$\hat{f}^{(2)}_{P,Q,g,\hat{a_2},h} : \{0,1\}^4 \to \{0,1\}^{160}$$

Repeating the above procedure grants us with a new pseudo-random $\vec{y} \in \{0,1\}^{2400}$ allowing us to define

$$\hat{f}^{(3)}_{P,Q,g,\hat{a_3},h} : \{0,1\}^{11} \to \{0,1\}^{160}$$

Using $f^{(3)}$ directly, all keys needed for our implementation can be manufactured.

As mentioned, the above process uses a random seed of size 2880, this can be improved by replacing the hash function h with a new hash function $\tilde{h}$ that has a smaller random key but still fulfills the requirements regarding h stated in the previous section. Therefore we define :

$$\tilde{h}_y(m) = (y_1 \hat{h}_{y_2}(m) + y_3 \hat{h}_{y_4}(m) + \hat{h}_{y_5}(m)) \bmod R$$

where $\hat{h}$ is the hash function defined in the previous section and $y = \langle y_1 \ldots y_5 \rangle$ is a uniformly distributed sequence of elements in $\mathbf{Z}_R$. Note that for any pair of different inputs x, y the collision probability, $\Pr_{\tilde{h}}[\tilde{h}(x) = \tilde{h}(y)]$, is extremely close to $1/R$ (at most $1/R + \left(\frac{13}{2^{160}}\right)^3$) and the key of $\tilde{h}$ is of size 800 bits. This new hash function $\tilde{h}$ was not used originally in the implementation of our pseudo-random function f because it is less efficient that h.

All in all the above construction with $\tilde{h}$ enables us to generate all random keys for the implementation of f using a random seed of 1600 bits.

4 Efficiency

1. In order to compute $\tilde{f}_{P,Q,g,\vec{a}}(x)$ for $|x| = n$ we need at most n multiplications modulo Q and one exponentiation modulo P.

2. Computing the value of $g^{2^i} \bmod P$ for $i = 0 \ldots \log Q$ as preprocessing improves our efficiency by turning the single modular exponentiation into $\log Q$ modular multiplications.

3. Additional preprocessing can improve the efficiency even further, for example computing the values of $g^{\alpha \cdot 4^i} \bmod P$ for all $\alpha = 1 \ldots 3$, $i = 0 \ldots \frac{\log Q}{2}$ will turn the single modular exponentiation into $\frac{\log Q}{2}$ modular multiplications, and computing the values of $a_{2i} \cdot a_{2i+1}$ for $i = 0 \ldots \frac{n}{2}$ will turn the n modular multiplications into $\frac{n}{2}$ modular multiplications. For more details see [1].

   In our implementation we preprocessed by computing $g^{\alpha \cdot 32^i} \bmod P$ for all $\alpha = 1 \ldots 31$, $i = 0 \ldots \frac{\log Q}{5}$. The reason for this specific choice is that, for technical reasons, our implementation performs part of its preprocessing on each run (and therefore there was no point in a more extensive preprocessing). In general, the more preprocessing done the faster the implementation will be.

4. The computation of f consists of two computations of $\tilde{f}$ (that are not entirely independent) and two hash executions. All in all computing f with our constants takes about 0.5 seconds. As mentioned above, for technical reasons our implementation performs part of the preprocessing on each run, making our running time about one second per execution.

5 Security

The construction we implemented is as secure as the decisional version of the Diffie-Hellman [2] assumption. For this to be true we need the keys to be random and secret. In our implementation we used pseudo-random keys that are as secure as the main construction (of f). Regarding the secrecy of the keys, since we lack the means for secret-'bulletproof' storage it might be feasible to access the keys.

References

[1] E. F. Brickell, D. M. Gordon, K. S. McCurley and D. B. Wilson, Fast exponentiation with precomputation, *Proc. Advances in Cryptology - EUROCRYPT '92*, LNCS, Springer, 1992, pp. 200–207.

[2] W. Diffie and M. Hellman, New directions in cryptography, *IEEE Trans. Inform. Theory*, vol. 22(6), 1976, pp. 644-654.

[3] M. Naor and O. Reingold, Number-theoretic constructions of efficient pseudo-random functions. To appear in: *Proc. 38th IEEE Symp. on Foundations of Computer Science*, 1997.

[4] V. Shoup, NTL package, http://www.cs.wisc.edu/∼shoup/ntl.