Public Key Encryption Algorithm and the Random Oracle

Divya Kumar Kala (05CS3021) & Varun Sharma (05CS1030)

April 15, 2009

1 Introduction

Cryptographic theory has provided a potentially invaluable notion for cryptographic practice: the idea of provable security. Unfortunately, theoretical work often gains provable security only at the cost of effciency. Schemes like standard RSA are efficient but not provably secured. There are schemes who are provably secured but are not efficient. However, there are encryption schemes, for eg. El Gamal encryption, which are efficient and secured at the same time. There is a need to construct more encryption schemes which are provably secure, but also efficient.One of the approach use is through Random Oracles.

2 Random Oracle

A Random Oracle is a theoretical black box that responds to every query with a (truly) random response chosen uniformly from its output domain, except that for any specific query, it responds the same way every time it receives that query i.e a random oracle is a mathematical function mapping every possible query to a random response from its output domain.

2.1 Random Oracle Model

The Random Oracle Model (ROM) was first formalised by Bellare and Rogaway. In the random oracle model, one assumes that some hash function is replaced by a publicly accessible random function (the random oracle). This means that the adversary cannot compute the result of the hash function by himself, he must query the random oracle. The random oracle model has been used to prove the security of numerous cryptosystems, and it has lead to simple and efficient designs that are widely used in practice. However, a proof in the random oracle model is not fully satisfactory, because such a proof does not imply that the scheme will remain secure when the random oracle is replaced by a concrete hash function (such as SHA-1). However, using a scheme that is proved in the Random Oracle model is better than no proof. However if at the slight cost of efficiency, we have a cryptosystem with a proof in the standard model (like that of the ElGamal encryption), then that is preferred.

2.1.1 Random Oracle Methodology

The Random Oracle Model, a popular methodology for designing cryptographic protocols consists of the following two steps. One first designs an ideal system in which all parties (including the adversary) have oracle access to a truly random function, and proves the security of this ideal system. Next, one replaces the random oracle by a good cryptographic hashing function (such as MD5 or SHA), providing all parties (including the adversary) with the succinct description of this function. Thus, one obtains an implementation of the ideal system in a "real-world" where random oracles do not exist.

2.1.2 Proof Technique

We assume that if an adversary A, has not queried for some point x, then $H(x)$ is completely random. We then try and construct a reduction, showing that if A is able to break the encryption scheme using the Random Oracle, then it can be used to break our standard cryptographic assumption. The reduction may choose value for the output of the Random Oracle and return to A. Also, it knows all the queries made to the Random Oracle.

2.1.3 Example

Consider a RSA based scheme,
- public key: $[N, e]$
- secret key: d
- Plaintext: $m \in \{0,1\}^{l(n)}$
- Enc: $< [r^e mod N, m \oplus H(r)] >$

If the RSA problem is hard and H is modeled as a Random Oracle, the construction has IND secured encryptions under CPA. Let A be a PPT, we define:

$$\epsilon(n) = Pr[PubK^{eav}_{A,\Pi}(n) = 1]$$

where $PubK^{eav}_{A,\Pi}(n)$ is defined as follows

1. A random function H is chosen.
2. Generate $< N, e, d >$. A is given $p_k =< N, e >$ and may query $H(.)$. Eventually A outputs two messages, $m_0, m_1 \leftarrow \{0,1\}^{l(n)}$
3. A random bit $b \leftarrow \{0,1\}$ and a random $r \leftarrow Z^*$ are chosen. A is given the ciphertext, $< [r^e modN, H(r) \oplus m_b] >$. The adversary can still query $H(.)$.
4. Finally, A outputs b'. $PubK^{eav}_{A,\Pi}(n)$ returns 1, if $b = b'$. Else 0 is returned.

We Define Query to be the event that at any point A queries r to the RO (where r is the value used to generate the challenge, c).

$$\begin{aligned} Pr[success] &= Pr[success \wedge \overline{Query}] + Pr[success \wedge Query] \\ &\leq Pr[sucess|\overline{Query}] + Pr[Query] \end{aligned}$$

Claim 1: $Pr[success|\overline{Query}] \leq 1/2$
Claim 2: $Pr[Query]$ is negligible

Claim 1 follows from the fact that if A does not query for r, then $H(r)$ is random, and so A has no way to understand whether m_0 or m_1 was encrypted.

For Claim 2
1. Construct a reduction D, which takes as input $c_1 = r^e modN$ and has to output r (i.e break RSA).
2. It generates randomly $c_2 \in \{0,1\}^{l(n)}$ and sends to A.
3. A makes some queries to $H, r_i.D$ observes the queries and checks if $r_i^e modN = c_1$.Whenever there is a match, thus RSA is broken. So, the $Pr[Query]$ must be negligible,under the standard RSA assumption.

3 Preliminaries

NOTATION, $\{0,1\}^n$ denotes the space of finite binary strings and $\{0,1\}^{infty}$ denotes the space of infnite ones. Strings are finite unless we say otherwise. We denote by $a||b$, or just ab, the string which is the concatenation of strings a and b. The empty string is denoted . A polynomial time algorithm is one which runs in time polynomial in its first argument. PPT stands for " probabilistic, polynomial time."A function $\epsilon(k)$ is negligible if for every c there exists a k_c such that $\epsilon(k) \leq k^{-c}$ for every $k \geq k_c$. A function is said to be non-negligible if it is not negligible. We'll use the notation "$k^{-\omega(1)}$ " to mean the class negligible functions or a particular anonymous function in this class.

Notation for probabilistic algorithms, spaces and experiments follows. If A is a probabilistic algorithm then, for any inputs $x, y, ...$ the notation $A(x, y, ...)$ refers to the probability space which to the string assigns the probability that A, on input $x, y, ...$,outputs σ . If S is a probability space we denote its support (the set of elements of positive probability) by $[S]$. If S is a probability space then $x \leftarrow S$ denotes the algorithm which assigns to x an element randomly selected according to S . In the case that S consists of only one element e we might also write $x \leftarrow e$. For probability spaces $S, T, ...$, the notation $Pr[x \leftarrow S; y \leftarrow T; .... : p(x, y, ...)]$ denotes the probability that the predicate $p(x, y, ...)$ is true after the (ordered) execution of the algorithms $x \leftarrow S, y \leftarrow T$, etc. Let f be a function. We extend this notation to define also probability spaces and algorithms via experiments. For example $\{x \leftarrow S; y \leftarrow T; .... : f(x, y, ...)\}$ denotes the probability space which to the string σ assigns the probability $Pr[x \leftarrow S; y \leftarrow T; .... : \sigma = f(x, y, ...)]$ And

$$< a, b, ... : x \leftarrow S; y \leftarrow T; .... : f(a, b, ..., x, y, ...) >$$

denotes the algorithm which on inputs $a, b, ...$ runs the sequence of experiments $x \leftarrow S, y \leftarrow T, ...$, and outputs $f(a, b, ..., x, y, ...)$.

ORACLES , For convenience, a random oracle R is a map from $\{0, 1\}^*$ to $\{0, 1\}^\infty$ chosen by selecting each bit of $R(x)$ uniformly and independently, for every x. Of course no actual protocol uses an infinitely long output, this just saves us from having to say how long "sufficiently long" is. We denote by 2^∞ the set of all random oracles.
The letter "R" will denote the "generic" random oracle, while $G : \{0, 1\}^* \leftarrow \{0, 1\}^\infty$ will denote a random generator and $H : \{0, 1\}^* \leftarrow \{0, 1\}^k$ a random hash function. Whenever there are multiple oracles mentioned, all of these are independently selected. Via all sorts of natural encodings, a single random oracle R can be used to provide as many independent random oracles as one wants.As usual the oracles provided to an algorithm are indicated by superscripts. Sometimes the oracle is understood and omitted from the notation.

Trapdoor Permutations. A trapdoor permutation generator is a PPT algorithm G_* which on input 1^k outputs (the encoding of) a triple of algorithms (f, f^{-1}, d). The rest two are deterministic and the last is probabilistic. We require that $[d(1^k)]$ be a subset of $\{0, 1\}^k$ and that f, f^{-1} be permutations on $[d(1^k)]$ which are inverses of one another. We require that there exist a polynomial p such that f, f^{-1} and d are computable in time $p(k)$, and that for all nonuniform polynomial time adversaries M ,

$$\epsilon(k) = Pr[(f, f^{-1}, d) \leftarrow G_*(1^k); x \leftarrow d(1^k); y \leftarrow f(x) : M(f, d, y) = x$$

is negligible. RSA 38 is a good example of trapdoor permutation. Call a trapdoor permutation generator G_* uniform if for all k and all $(f, f^{-1}, d) \in [G(1^k)]$ it is the case that d is the uniform distribution on $\{0; 1\}^k$.

Encryption. We extend the notion of public key encryption to the random oracle model.The scheme is specified by a PPT generator G which takes a security parameter 1^k and outputs a pair of probabilistic algorithms (E, D) which are called the encryption and decryption algorithms respectively and which run in time bounded by $G's$ time complexity. A user U runs G to get (E, D) and makes the former public while keeping the latter secret. To encrypt message x anyone can compute $y \leftarrow E^R(x)$ and send it to U ; to decrypt ciphertext y user U computes $x \leftarrow D^R(y)$. We require $D^R(E^R(x)) = x$ for all x and assume for simplicity that $D^R(y) = 0$ if y is not the encryption under E^R of any string x.

4 Security in the Random Oracle Model

Definition. We adapt the notion of polynomial security to the random oracle model. A CP-adversary (chosen-plaintext adversary) A is a pair of nonuniform polynomial time algorithms $(F; A_1)$, each with access to an oracle. For an encryption scheme G to be secure in the random oracle model we require that for any CP-adversary $A = (F, A_1)$,

$$Pr[R \leftarrow 2^\infty; (E, D) \leftarrow G(1^k); (m_0, m_1) \leftarrow F^R(E); b \leftarrow \{0, 1\};$$
$$\alpha \leftarrow E^R(m_b) : A_1^R(E, m0, m1, \alpha) = b] \leq \tfrac{1}{2} + k^{-\omega(1)}.$$

Note that the oracle used to encrypt and decrypt is given to the adversary who tries to distinguish the encryption of strings m_0 and m_1, so, for example, a hash $H(x)$ with H derived from R could most certainly not appear in the secure encryption of a string x.

Encryption By $E(x) = f(r)||G(r) \oplus x$ .To specify our encryption scheme, let G_* be a trapdoor permutation generator and let $G : \{0, 1\}^* \rightarrow \{0, 1\}^\infty$ be a random generator. On input 1^k our generator G runs G_* to get (f, f^{-1}, d) . It sets E^G to the following algorithm:

$$E^G \leftarrow< x : r \leftarrow d(1^k) : f(r)||G(r) \oplus x >$$

where $G(r) \oplus x$ denotes the XOR of the first $|x|$ bits of $G(r)$ with x. Of course the decryption function is then $D^G(ys) = s \oplus G(f^{-1}(y))$.

4.1 Theorem : The $E(x) = T(r)||G(r) \oplus x$ Scheme Is Secure Against Chosen Plaintext attack in the Random Oracle model for trapdoor T.

We will prove this by contradiction. Suppose this is not true. That is we have an adversary $A = (A_0, A_1)$ with significant advantage .A is used to generate the plaintexts m_0 and m_1. A_1 is then handed the challenge c, which is the ciphertext corresponding to a randomly chosen message.Both A_0 and A_1 can make queries to the random oracle G. Using these algorithms we intend to invert T, the trap-door function without knowing the trap-door.
If A_0 asks a query for r (used to generate the challenge), return r (thus we have inverted the trap-door). Else A_0 terminates,and A_1 starts.
Instead of feeding A_1 the challenge ciphertext, it is asked $T(r)||z$ where $z = \{0,1\}^{|x|}$ is a random string. It is checked whether A_1 makes a query at r, by checking if $T(r) = y$.
Define A_k : Event that A_1 asks a query at r. If it does not then it has no advantage in guessing which plaintext was encrypted.

$$\begin{aligned} 1/2 + \epsilon &< Pr[Asucceeds|A_k]Pr[A_k] + Pr[Asucceeds|\overline{A_k}]Pr[\overline{A_k}]. \\ &< Pr[A_k] + 1/2 \end{aligned}$$

Thus $Pr[A_k] > \epsilon$ Thus we can invert the trapdoor T with significant probability,thus we arrive at a contradiction.

4.2 Theorem : The $E(x) = f(r)||G(r) \oplus x||H(rx)$ Scheme Is Secure Against Chosen Ciphertext attack.

Let $A = (F, A_1)$ be an RS-adversary that succeeds with probability $\frac{1}{2} + \epsilon(k)$ for some non negligible function $\epsilon(k)$. We construct an algorithm $M(f, d, y)$ that computes $f^{-1}(y)$ non-negligibly often, where $(f, f^{-1}, d) \leftarrow G_*(1^k); r \leftarrow d(1^k); y \leftarrow d(r)$. Algorithm M begins by running $F(E)$ where E is defined from f as specified by our scheme. F takes three oracles, namely, $G, HandD^{G,H}$, whose queries are answered by F as follows. If a query r to G satisfies $f(r) = y$ then M outputs r and halts; else it returns a random

string of the appropriate length. If a query rx to H satisfies $f(r) = y$ then M outputs r and halts; else it returns a random string of the appropriate length.To answer query $a||w||b$ to $D^{G,H}$, algorithm M sees if it has already asked some query r of G and ru of H, where $a = f(r)$ and $w = G(r) \oplus u$, and if so returns u; else it returns invalid.If M completes the running of $F(E)$ then it obtains an output (m_0, m_1). Now M runs $A_1(E, m_0, m_1, \alpha)$ where $\alpha = y||w||b$ for $w \leftarrow \{0,1\}^{|m_0|}$ and $b = \{0,1\}^k$. Once again, M must simulate the behavior of queries to $G, H, and D^{G,H}$. This is done exactly as before, when F was being run by M.
To see that this construction works, first consider the "$real$" environment of A running with its oracles. Let A_k denote the event that $a||w||b \leftarrow F(E)$, for some $a, w, and b$, and A made some oracle call of $G(r) or H(ru)$, where $f(r) = a$.Let L_k denote the event that A_1 asks $D^{G,H}$ some query $a||w||b$ where $b = H(f^{-1}(a)||w \oplus G(f^{-1}(a)))$, but A_1 never asked its $H - oracle$ for the image of $f^{-1}(a)||w \oplus G(f^{-1}(a))$. Let $n(k)$ denote the total number of oracle queries made. It is easy to verify that $Pr[Lk] \leq n(k)2^{-k}$. It is also easy to see that

$$Pr[Asucceeds|\overline{L_k} \wedge \overline{A_k}] = \tfrac{1}{2}.$$

Thus $\frac{1}{2} + \epsilon(k) = Pr[Asucceeds]$ is bounded above by

$$Pr[Asucceeds|Lk]Pr[Lk] + Pr[Asucceeds|\overline{L_k} \wedge A_k]Pr[\overline{L_k} \wedge A_k] +$$
$$Pr[Asucceeds|\overline{L_k} \wedge \overline{A_k}]Pr[\overline{L_k} \wedge \overline{A_k}]$$

which is at most $n(k)2^{-k} + Pr[A_k] + \frac{1}{2}$. And so

$$Pr[Ak] \geq \epsilon(k) - n(k)2^{-k}.$$

Now, returning to the simulation of A by M, note that M fails to behave like A with probability bounded by $Pr[Lk]$, and so

$$Pr[Minvertsfaty] \geq \epsilon(k) - n(k)2^{-k+1}$$

which is still non negligible. This completes the proof.